COS 222 - Semester Test 2 Notes

Regan Koopmans

October 22, 2016

Contents

1 Chapter 7 - Memory Management 3
1.1 Memory Management Requirements 3
1.1.1 Relocation 3
1.1.2 Protection 3
1.1.3 Sharing 4
1.1.4 Logical Organization 4
1.1.5 Physical Organization 4
1.2 Memory Partitioning 4
1.2.1 Fixed Partitioning 4
1.2.2 Dynamic Partitioning 5
1.2.3 The Buddy System 5
1.2.4 Relocation 6
1.3 Paging 6
1.4 Segmentation 6
1.5 Loading And Linking 7
1.5.1 Loading 7
1.5.2 Linking 7

2 Chapter 8 - Virtual Memory 8
2.1 Hardware Control Structures 8
2.1.1 Locality and Virtual Memory 8
2.1.2 Paging 8
2.1.3 Segmentation 8
2.1.4 Combined Paging and Segmentation 8
2.1.5 Protection and Sharing 8
2.2 Operating System Software 8
2.2.1 Fetch Policy 9

2.2.2 Placement Policy 9
2.2.3 Replacement Policy 9
2.2.4 Resident Set Management 10
2.2.5 Cleaning Policy 10
2.2.6 Load Control 11
2.3 Unix and Solaris Memory Management 12
2.3.1 Paging System 12
2.3.2 Kernel Memory Allocator 12
2.4 Linux Memory Management 12
2.4.1 Linux Virtual Memory 12
2.5 Windows Memory Management 12
2.5.1 Windows Virtual Address Map 12
2.5.2 Windows Paging 12
2.5.3 Windows 8 Swapping 12
2.6 Android Memory Management 12
2.6.1 ASHMem 12
2.6.2 Pmem 12
2.6.3 Low Memory Killer (. . . wat) 12

3 Chpater 9 - Uniprocessor Scheduling 13
3.1 Types of Scheduling 13
3.1.1 Long-Term Scheduling 13
3.1.2 Medium-Term Scheduling 13
3.1.3 Short-Term Scheduling 13
3.2 Scheduling Algorithms 13
3.2.1 Short-Term Scheduling Criteria 13
3.2.2 The Use of Priorities 13
3.2.3 Alternative Scheduling Policies 13
3.2.4 Performance Comparison 13
3.2.5 Fair-Share Scheduling 13
3.3 Traditional UNIX Scheduling 13

4 Chapter 10 - Multiprocessor Scheduling 13
4.1 Multiprocessor and Multicore Scheduling 13
4.1.1 Granularity 14
4.1.2 Design Issues 14
4.1.3 Process Scheduling 14
4.1.4 Thread Scheduling 14
4.1.5 Multicore Thread Scheduling 15
4.2 Real-Time Scheduling 15

4.2.1 Characteristics of A Real Time Operating System . . . 15
4.2.2 Real-Time Scheduling 16
4.2.3 Deadline Scheduling 16
4.2.4 Rate Monotic Scheduling 16
4.2.5 Priority Inversion 17
4.3 Linux Scheduling 17
4.3.1 Real-Time Scheduling 17
4.3.2 Non-Real-Time Scheduling 17
4.4 UNIX SVR4 Scheduling 17
4.5 Unix FreeBSD Scheduling 17
4.5.1 Priority Classes 17
4.5.2 SMP and Multicore Support 17
4.6 Windows Scheduling 17
4.6.1 Process and Thread Priorities 17
4.6.2 Multiprocessor Schduling 17

1 Chapter 7 - Memory Management

1.1 Memory Management Requirements

A **Frame** is a fixed-length block of main memory. A **Page** is a fixed-length block of data that resides in secondary memory (such as a disk). A page may temporarily be copied into a frame of main memory. A **Segment** is a variable-length block of data that resides in secondary memory. An entire segment may be temporary copied into an available region of main memory (this process is known as segmentation) or the segment may be divided into pages which can individually be copied into main memory (combined segmentation and paging).

1.1.1 Relocation

We need our system to be able to move the location where a process exists, particularly in the case where we might want to swap it for another process.

1.1.2 Protection

Each process should protected against unwanted interference by other processes, whether accidental or intentional. Thus programs in other processes should not be able to reference memory loctions in a process for reading or writing without sufficient permission.

1.1.3 Sharing

In order to minimize redundancy, we would ideally like processes to be able to share blocks of memory, otherwise there will be numerous instances where this data would have to be duplicated.

1.1.4 Logical Organization

The main memory of a computer system is most likely to be arranged as a linear, or one-dimensional, address space of either bytes or words.

1.1.5 Physical Organization

Memory is organised in at least two levels, namely primary and secondary memory

The main memory available for a program plus its data may be insufficient. In that case, the programmer must engage in a practise known as **overlaying**.

1.2 Memory Partitioning

1.2.1 Fixed Partitioning

This is the idea that memory is split into certain unchanging sizes that may be allocated to a process. Fixed partitioning is almost unknown in modern times because of the inherent memory wastage and inflexibility it introduces into the system.

1. Partition Sizes

   Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. A program will be forced to occupy some partition that is larger than its maximum size. This is wasted space, and we call this waste **internal fragmentation**.

2. Placement Algorithm

   With `equal partition` sizes the pacement of processes in memory is trivial. As long as there is any available partition, a process can be loaded into that partition. Since all partition sizes are equal, there is no benefit in choosing one free partition over another.

   With `unequal partition` sizes there are **two** possible ways to assign processes to partitions:

The simplest way is to assign the process to **the smallest partition that fits**. In this case we will need a queue for each partition. The benefit of this is that we are minimizing internal fragmentation and therefore minimizing memory wastage.

Alternatively, we can use a single queue for all processes. A process will be allocated the next free block that is closest to its requiremens. If main memory can no longer support any new processes, then a swapping decision must take place, and this becomes more the job of the scheduler.

1.2.2 Dynamic Partitioning

With dynamic partitioning, the partitions are of variable length and number. Dynamic partitioning suffers from external fragmentation, in which processes are scattered across main mememory, with gaps inbetween, each of which too small to accomodate another processes. To overcome this we would need to perform **compaction**, which is similar to defragmentation on a harddrive.

1. Placement Algorithm

   (a) Best Fit
   This algorithm chooses the block that is closes to that of the request.

   (b) First Fit
   This algorithm chooses the block that first satisfies the space requirement

   (c) Next-fit
   This algorithm chooses the second available block for the memory.

2. Replacement Algorithm

   `These are discussed in virtual memory`

1.2.3 The Buddy System

Both fixed and dynamic partitioning schemes have their drawbacks. A fixed partitioning scheme limits the number of active processes and may use space inefficiently.A dynamic partitioning scheme bears the overhead of compaction.

In a buddy system, the main memory is always broken into chunks of the power off 2. The main memory will be halved until halving it would make it too small for the process. When enough processes have finished the system may be able to merge these partitions back together.

[512] [256][256]

1.2.4 Relocation

A **logical address** is a reference to a memory address that is independent of the current assignment of data to memory, and therefore a translation is requred if we want to use the reall addresses these point to.

Relative Addresses are addresses that express locations in memory in terms of a base address and an offset. This base address may be the start of a segment or really any arbitrary label.

Physical Addresses are absolute addresses within main memory.

If the process control block maintains a single base pointer then all addresses can be relative to this pointer. This base pointer can be modified, and in this way a process can maintain all of its logical addresses, while simulaniously swapping in and out of different locations in memory.

1.3 Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory; the former results in **internal fragmentation** and the latter in **external fragmentation**.

Suppose, however, that memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-sized chunks of the same size. The chunks of the process are known as **pages**, which can be assigned into chunks of memory, **frames**.

1.4 Segmentation

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments**. It is not required that all segments of all programs be of the same length, although there is a maximum segment length.

The difference between segmentation and paging is that:

Paging	Segmentation
Transparent to programmer	Involves the programmer
No seperate protection	Offers protetion
No seperate compiling	
No shared code.	Shared program

1.5 Loading And Linking

1.5.1 Loading

Loading is essentially bringing a process into main memory such that is can run.

1. Absolute Loading

   An absolute loade requires that a given load module always be loaded into the same locations in main memory. Thus, in the load module presented to the loader, all adress references must be to specific, or "absolute", main memory addresses.

2. Relocatable Loading

   The disadvantage of binding memory references to specific addresses prior to loading is that the resulting load moddule can only be placed in one region of main memory. However, when many programs share main memory, it may not be desirable to decide ahead of time into which region of memory a particular module should be loaded. It is better to make that decision at load time.

3. Dynamic Run-time Loading

   Relocatable loaders are common and provide obvious benefits relative to absolute loaders. However, in a multiproramming environment, even one that does not depend on virtual memory, the relocatable loading scheme is inadequate.

1.5.2 Linking

The function of a linker is to take as input a collection of object modules and produce a load module, consisting of an integrated set of prgram and data modules, to pass to the loader.

1. Linkage Editor

   The nature of thos address linkage will depend on the type of module to be created and when the linkage occurs. If, as is usually the case, a relocatable load module is desired,

2. Dynamic Linker

   As with loading, is is possible to defer some linkage functions. The term **dynamic linking** is used to refer to the practise of deferring the linkage of some external modules until after

For **load-time dynamic linking** the following step occur:

(a)

2 Chapter 8 - Virtual Memory

2.1 Hardware Control Structures

2.1.1 Locality and Virtual Memory

2.1.2 Paging

1. Page Table Structure
2. Inverted Page Table

   The inverted page table (IPT) is best thought of as an off-chip extension of the TLB, which uses normal system RAM. Unlike the true page table, it is not necessarily able to hold all current mappings.

   The inverted page table allows processes to potentiall share pages.

2.1.3 Segmentation

1. Virtual Memory Implications

2.1.4 Combined Paging and Segmentation

Both paging and segmentation have their strengths. Paging, which is transparent to the programmer, eliminates external fragmentation and thus provides efficient use of memory. Segmentation, which is visible to the porgrammer has the benefit of maintaining growing data structures, modularity and support for sharing and protection.

2.1.5 Protection and Sharing

2.2 Operating System Software

The design of the memory management portion of an OS depends on three fundamental areas of choice:

- Whether or not to use virtual memory techniques
- The use of paging, segmentation or both.
- The algorithms employed for various aspects of memory management.

2.2.1 Fetch Policy

1. Demand Paging

   This is the more simple of the two fetching policies. Pages are brought into memory when they are requested, which effectively means that a page fault occurs. This means that at the beginning of the systems' run-time, page faults will be numerous, but will decrease as the popular pages get proceedurally added to main memory.

2. Prepaging

   This policy attempts to predict the realistic future page use, usually by means of the **Principle of Locality**. Rather than simply retrieving one page, it retrieves a certain amount of its neighbours.

2.2.2 Placement Policy

2.2.3 Replacement Policy

When the memory we have available to load pages becomes full, we need certain heuristics that allow us to logically replace and evict certain pages.

1. Frame Locking

   One restriction on replacement policy needs to be mentioned before looking at algorithms. Some of the frames in main memory may be **locked**. Essential frames such as those that the kernel resides in, or I/O buffers, are locked and hence cannot be replaced.

2. Basic Algorithms

   (a) Optimal
   The optimal replacement policy is a theoretical concept that could only be implemented with perfect information about the past, present and future of the system.

   (b) Least Recently Used (LRU)
   In this replacement policy (which happens to be one of the most popular).

   (c) First-in-First-Out (FIFO)
   This replacement policy will pereferencially remove older pages to newer ones.

    (d) Clock

        This replacement policy is a circular list

3. Page Buffering

   An interesting strategy that can improve paging performance and allow the use of a simpler paging replacement policy is that of page buffering.

2.2.4 Resident Set Management

The resident set of a process is the pages of that process that currently reside in main memory.

1. Resident Set Size

   The smaller the memory allocated to each process is, the more processes can reside in main memory, and hence it is more likely that the operating system can find a process that is ready to run.

2. Replacement Scope

   The scope of a replacement strategy can be

   (a) Local Replacement Strategy
       i. Fixed Allocation
       ii. Variable Allocation
   (b) Global Replacement Strategy
       i. Variable Allocation

2.2.5 Cleaning Policy

These are the policies used to decide which pages should be removed from main memory. These poilicies mirror/complement the fetching policies. Cleaning policies are important.

1. Demand

   With demand cleaning, a page is written out to secondary memory only when it has been selected for replacement.

2. Precleaning

   The precleaning policy will write to seconday memory early such that pages can be expelled from main memory in batches.

2.2.6 Load Control

Load control is concerned with determining the number of processes that will be resident in main memory, which has been referred to as the **multi-programming level**.

1. Multiprogramming Level

   As the multiprogramming level increases, so does thrashing, and at some critical point we achieve **livelock**, where processes spend their entire processing time, or large portions of it, swapping into and out of main memory.

2. Process Suspension

   If the degree of multiprogramming is to be reduced, we need some heuristics to systematically suspend certain processes. Here are a few measures to consider:

   (a) Lowest-priority process
   (b) Fauling process
   (c) Last process activated
   (d) Process with the smallest resident set
   (e) Largest process
   (f) Process with largest remaining execution window

2.3 Unix and Solaris Memory Management

2.3.1 Paging System

2.3.2 Kernel Memory Allocator

2.4 Linux Memory Management

2.4.1 Linux Virtual Memory

2.5 Windows Memory Management

2.5.1 Windows Virtual Address Map

2.5.2 Windows Paging

2.5.3 Windows 8 Swapping

2.6 Android Memory Management

Android include s a number of extensions to the normal Linux kernel memory management facility

2.6.1 ASHMem

This feature provides anonymous shared memory, which abstracts memory as file descriptors. This file descriptor can then be parsed to another process to use.

2.6.2 Pmem

This feature allocates virtual memory so that is it contiguous in memory. This is especially useful for devices that do not explicitely support virtual memory.

2.6.3 Low Memory Killer (. . . wat)

Most mobile devices do not have swap capabilities. This memory feature allows the Android operating system to warn apps to lower their memory usage. If an app is unable to, or does not comply, it is terminated.

3 Chpater 9 - Uniprocessor Scheduling

3.1 Types of Scheduling

3.1.1 Long-Term Scheduling

The long-term scheduler determines which programs are admitted into main memory.

3.1.2 Medium-Term Scheduling

Medium-term sheculing is part of the swapping function. This scheduling is the decision of what processes should be partiallly or fully in main memory.

3.1.3 Short-Term Scheduling

Also known as the **dispatcher**, short-term scheduling involves deciding what process should be executed next by the processor.

3.2 Scheduling Algorithms

3.2.1 Short-Term Scheduling Criteria

3.2.2 The Use of Priorities

In many systems, each process is assigned a priority and the scheduler will always choose a process of higher priority over one with lower priority.

3.2.3 Alternative Scheduling Policies

3.2.4 Performance Comparison

3.2.5 Fair-Share Scheduling

3.3 Traditional UNIX Scheduling

4 Chapter 10 - Multiprocessor Scheduling

4.1 Multiprocessor and Multicore Scheduling

We can classify multiprocessor systems as follows

- Loosely coupled
- Functionally specialized
- Tightly coupled multiprocessor

4.1.1 Granularity

Grain Size	Description	Synchronization Interv
Fine	Parallelism inherent in single execution stream	< 20
Medium	Parallelism processing or multitasking in single application	20 - 200
Coarse	Distributed of concurrent processes in multiprogramming env	200 - 2000
Very Coarse	Distributed processing across network nodes	2000 - 1 million
Independent	Multiple unrelated processes	Not applicable

4.1.2 Design Issues

1. The Assignment of Processes to Processors

   If a process is permanently assigned to one processor from its activation to completion

2. THe use of multiprogramming on individual processors

3. Actual Dispatching of processes.

4.1.3 Process Scheduling

In most traditional multiprocessor systems, processes are not dedicated to processors. Rather, there is a single queue for all processors, or if some sort of priority scheme used, multiple queues representing priority, all feeding to a common pool of porcessors

4.1.4 Thread Scheduling

There are four main approaches to scheduling threads, namely:

1. Load Scheduling

   Processes are not assigned to a particular processor. A global queue of ready threads is maintained, and each processor, when idle, selects a thread from the queue. Some Load Scheduling algorithms include:

   (a) FCFS
   (b) Smallest number of threads first
   (c) Preemptive smallest number of threads first

2. Gang Sheduling

   A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.

3. Dedicated Processor Assignment

4. Dynamic Scheduling

   The number of threads in a proess can be altered during the course of execution.

4.1.5 Multicore Thread Scheduling

4.2 Real-Time Scheduling

A **hard real-time task** is one that must be met by the deadline, otherwise it may cause unacceptable or fatal damage to the systems' function. An example of such a task is creating s Process Control Block for some essential process.

A **soft real-time task** is one in which we would like to complete in time, but we do not expect the entire system to fail if we miss the deadline slightly.

Another important concept for real time scheduling is that of **periodic** and **aperiodic** tasks. These will affect our decisions, as we will want to keep pages related to periodic tasks in main memory as much as possible.

4.2.1 Characteristics of A Real Time Operating System

Real time operating systems have unique requirements in the floowing general areas

- **Determinism**
- **Responsiveness**
- **User control**
- **Reliability**
- **Fail-soft operation**

4.2.2 Real-Time Scheduling

These are the classes of real time scheduling algorithms:

- Static table-driven approaches
- Static priority-driven preemptive approaches
- Dynamic planning-based approaches
- Dynamic best effort approaches

4.2.3 Deadline Scheduling

Most contemporary real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible, and hence emphasize rapid interrupt handling and task dispatching.

Extra information includes:

- **Ready Time**
- **Starting Deadline**
- **Completion Deadline**
- **Processing Time**
- **Resource Requirements**
- **Priority**
- **Subtask Structure**

4.2.4 Rate Monotic Scheduling

Rate monotic scheduling is a scheduling algorithm used in real -time scheduling systems, using a static priority scheme. Priority is based on cycle duration of the job, and therefore shorter cylces implies higher priority. Rate monotic analysis is used in conjunction with these systems to provide scheduling guarantees for a particular application.

4.2.5 Priority Inversion

Priority inversion is the event where a high priority process is forced to wait for a lower priority process. This occurs because the lower priority process is holding some resource that the higher priority process requires.

In practical terms, two alternative approaches are used to avoid unbounded priority inversion: **Priority inheritence** and **Priority ceiling protocol**.

4.3 Linux Scheduling

4.3.1 Real-Time Scheduling

The three linux scheduling classes are as follows:

- $\text{SCHED}_{\text{FIFO}}$
- SCHED_{RR} (round robin)
- $\text{SCHED}_{\text{OTHER}}$

4.3.2 Non-Real-Time Scheduling

4.4 UNIX SVR4 Scheduling

4.5 Unix FreeBSD Scheduling

4.5.1 Priority Classes

4.5.2 SMP and Multicore Support

4.6 Windows Scheduling

4.6.1 Process and Thread Priorities

4.6.2 Multiprocessor Schduling