SE 350 End of Term Notes

Craig, Shale
sakcraig@uwaterloo.ca

February 25, 2014

Contents

Chapter 1

Memory

1.1 Paging/Segmentation

1.1.1 Paging

Paging allows memory to be comprised of fixed-size blocks that are addressed by virtual addresses that are page numbers and an offset. Each page can be anywhere in main memory.

1.1.2 Address Translation

Address translation (logical, not physical addresses) allow us to use non-contiguous memory layouts, which allows processes to run without being fully resident in memory.

Execute code. Once program tries to read/exec instructions not in RAM, we page fault, block, read data, then resume.

Virtual Addresses are tuples of $\langle \text{Page \#}, \text{Offset} \rangle$. We look up the Page Number + Page Table Pointer in the Page Table, which gives us the Frame #. Combine (by a bitmask) the two points, and you get the physical address of the memory in main memory.

Address Translation (i.e. a root page table) allows us to have large page tables, and keep some of the page tables in main memory while they are not being accessed.

The downside of Page Tables is that Page Table size is proportional to the virtual address space.

1.1.3 Segmentation

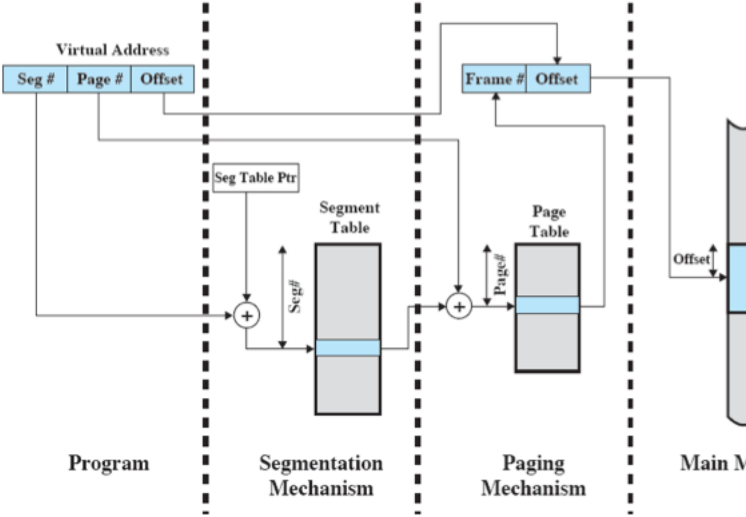
Allows the programmer to view memory as multiple address spaces or segments. We can now share data among processes, and protect segments from data modification.

Segmentation is pretty much the same as virtual addressing, except the segment table contains the base address and is added to the offset.

1.1.4 Combined Paging and Segmentation

Paging is transparent to the programmer while Segmentation is visible to the programmer.

See figure 1.1.4 for more information.



1.2 Replacement Strategies

- Least Recently Used
- First-in, First-Out
- Clock Policy
- Page Buffering

1.2.1 Page Buffering

Memory pages are cached and are placed into a “free page” or “modified” page list if they have or haven’t been modified respectively. This is done so the OS can revive these pages from the list if space becomes available.

1.3 Page Size v.s. Page Faults

By having an smaller page size, all parts of the pages in memory will be relevant to the process in recent references. If they are bigger, there will be “useless” portions that aren’t used

1.4 Working Set v.s. Resident Set

Resident set is the portion of a process that is in main memory. The smaller the resident set size, the higher number of processes that can be in memory. Once it is past a certain size, there is no real gain from a large resident set.

The working set is the set of pages of the process that have been referenced in the last t time.

1.5 Calculating the Resident Set Size

Variable allocation means the size of the working set for one process is fixed with respect to time. Variable allocation means the size of the working set for one process varies with respect to time.

There are three main types of allocation strategies:

- Fixed Allocation, Local Scope:

  Decide before how big the working set should be, then execute under that decision.

- Variable Allocation, Local Scope:

  New processes get a working set size based on a heuristic. Page faults result in pages from the current (local) process's working set being kicked out.

- Variable Allocation, Global Scope:

  New processes get a working set size based on a heuristic. Page faults result in pages from any (i.e. global) process's working set being kicked out.

1.6 Principle of Locality

Stuff you need in the future is close to stuff you needed in the past.

Chapter 2

Simultaneous Execution

2.1 Preconditions for Deadlock

Preconditions for deadlock are as follows:

- Mutual Exclusion (i.e. no way to ensure processes use resources one at a time)
- Hold-And-Wait
- No preemption (with respect to resources)
- Circular wait

2.2 Semaphores, Monitors, etc

2.2.1 Mutexes

Special machine instructions allow us to test and set a variable in a single machine instruction (atomically).

```
boolean testSet(int i):
    if (i == 0):
        i = 1
        return true
    else:
        return false

void exchange(int register, int mem
    temp = memory
    memory = register
    register = temp
```

This is simple and applicable to any number of processes on single or multiple processors, but it does busy-waiting and can allow starvation if there are multiple waiting processes.

2.2.2 Semaphores

Semaphores are special variables that are used for signaling. Semaphores are initialized to a nonnegative number, generally the maximum number of concurrent accesses. “semWait” decrements the value, “semSignal” increments the semaphore value.

```
void semWait(semaphore s):
    s.count--
    if (s.count < 0):
        s.queue.push(getCurrentProc

void semSignal(semaphore s):
    s.count++
    if (s.count <= 0):
        p = s.queue.pop()
        p.makeReady()
```

Binary semaphores are the same, but they only take on binary values.

```
void semWaitB(binary_semaphore s):
    if (s.value == 1):
        s.value = 0
    else:
        s.queue.push(getCurrentProc

void semSignalB(binary_semaphore s
    if (s.queue.isEmpty()):
        s.value = 1
    else:
        p = s.queue.pop()
```

```
                    p.makeReady()
```

2.3 Amdhal's Law

As the level of Multiprogramming increases, the returns will become asymptotically faster, but the limit will be constant. This is because there is a limited set of instructions that can be run at the same time.

2.4 Scheduling Algorithms

- Rate Monotonic

  Lower bound on schedulable utilization = 0.693. Highe priority task is the one with the shortest period.

- Earliest Deadline First:

  Can schedule a CPU utilization of 1. Highest-priority task is the one with the next deadline.

Chapter 3

Other

3.1 States for Processes

- New
- Running
- Ready
- Blocked
- Ready-suspend
- Blocked-suspend
- Exit