CSCI 418—Operating Systems

Lecture 2

Memory Management–early systems

Textbook: Operating Systems — Internals and Design Principles (9th edition)
by William Stallings

1. A brief history of OS development

- **1. First generation (1940–1955)**, mainly used in military.
- **2. Second generation (1955–1965)**, mainly used in business.
- **3. Third generation (1960s–late 1970s)**.
- **4. Post-3rd generation (late 1970s–early 1990s)**.
- **5. Modern generation (mid-1990s–now)**.

2. Types of OS

- **1. Batch system**. Example: Systems processing punched cards, tapes, etc.
- **2. Interactive system**. Example: DOS running on a PC.
- **3. Real-time system**. Example: High speed aircraft, cruise missile.
- **4. Hybrid system**. Example: Combination of batch and interactive system, e.g., CM-5.
- **5. OS for intelligent phone**. Example: Android.
- **6. Embedded system**. Example: Kernel for a robot, elevators.

3. Early Memory Management Systems

- In the early days, a computer can only have one user at one time. Moreover, a computer can only run a program at a time. To run a program, it must be entirely and contiguously loaded into memory. The memory management is therefore easy.
- **Algorithm**: Load a job in a single-user system

```
– 1.  Store first memory location Y of program into
      base register
– 2.  Set program counter = Y
– 3.  Read first instruction of program
– 4.  Increment program counter by # of bytes of
      instruction
– 5.  Last instruction?
–       if YES, then stop loading
–       if NO, then continue with step 6
– 6.  Program counter > memory size?
–       if YES, then stop loading
–       if NO, then continue with step 7
– 7.  Load instruction to memory
– 8.  Read next instruction of program
– 9.  Go to step 4
```

4. Fixed (Static) Partitions

- Single-user system cannot support **multiprogramming**, which is especially not cost-effective in the business community.
- Static partition is one way to handle multiprogramming.
- Once the system is power on and reconfigured, the partition sizes remain static. Partition sizes can only be changed/reconfigured when computer is rebooted.
- Any program must be entirely and contiguously stored in a partition.
- Clearly, several programs (jobs) can reside in memory at the same time.
- What is the drawback of fixed partition?

- **Algorithm to load a job in a fixed partition**

```
– 1.  Determine job's requested memory size.
– 2.  If job_size > size of largest partition
–       then reject the job
–          print appropriate message to operator
–          go to step 1 to handle the next job
–       else continue step 3.
– 3.  Set i = 1.  //i is the counter
– 4.  While i <= number of partitions in memory
–       if job_size > memory_partition_size(i)
–          then i = i +1
–       else
–          if memory_partition_status(i)='FREE'
–             then load job into memory_partition(i)
–                  change memory_partition_status(i)
–                  to 'BUSY'
–                  go to step 1
–          else i = i + 1
– 5.  No partition available at this time, put job
–     in waiting queue.
– 6.  Go to step 1 to handle next job.
```

5. Dynamic Partitions

- There is no partition when the computer is turned on.
- Available memory is still kept in contiguous blocks but jobs are given only as much memory as they request.
- It still does not solve the memory-wasting problem completely.

- **External Fragmentation**: the dynamic allocation of memory creates fragments of free memory between allocated memory, which might be wasted.
- Notice that when memory is allocated, we can either use **first-fit** (first partition fitting the requirements) or **best-fit** (closest fit, the smallest partition fitting the requirements).
- The **first-fit** allocation method keeps a list of free/busy memory fragments ordered by memory address.
- The **best-fit** allocation method keeps a list of free/busy memory fragments ordered by memory size.
- What is the advantage/drawback of first-fit?
- What is the advantage/drawback of best-fit?

- **Algorithm: Dynamic partition–first fit**

```
– 1.  i = 1.  //i = counter
– 2.  While i<= number of blocks in memory
–       if job_size > memory_size(i)
–          then i = i + 1.
–       else
–          load job into memory_size(i)
–          adjust free/busy memory lists
–          go to step 4.
– 3.  Put job in waiting queue.
– 4.  Process next job.
```

- **Algorithm: Dynamic partition–best fit**

```
– 1.  memory_block(0)=9999999.  //largest memory
  address
– 2.  initial_memory_waste=memory_block(0)-job_size.
– 3.  j = 0.  //j is the subscript.
– 4.  i = 1.  //i is the counter.
– 5.  While i<= number of blocks in memory
–       if job_size > memory_size(i)
–          then i = i + 1.
–       else
–          memory_waste=memory_size(i)-job_size
–          if initial_memory_waste > memory_waste
–            then j = i
–                 initial_memory_waste = memory_waste
–            i = i + 1.
– 6.  If j==0 then put job in waiting queue
–            else load job into memory_size(j)
–                 adjust free/busy memory lists
– 7.  Process next job.
```

6. Memory Deallocation

- When memory spaces are not used anymore they must be released (returned back to) the system.
- Fixed partition: easy! just reset the 'free/busy' status variable.
- Dynamic partition: we need to combine free areas of memory.
- Think of 3 situations when a block is to be released:

- **Algorithm: Dynamic partition–memory deallocation**

```
– 1.  If job_location is adjacent to one or more
      free blocks
–     then
–       if job_location is between 2 free blocks
–          then merge all three blocks into one.
–               memory_size(i-1)=memory_size(i-1)
–                    +job_size+memory_size(i+1).
–               set the status of memory_size(i+1)
–               to NULL.
–          else merge both blocks into one
–               memory_size(i-1)=memory_size(i-1)
–                             +job_size.
–     else search for NULL entry in free memory
–          list.
–          enter job_size and beginning_address
–          in the entry list.
–          set its status to FREE.
```

7. Relocatable Dynamic Partitions

- Both of the previous partitions introduce internal/external fragmentations.
- Relocatable dynamic partition is a natural solution.
- The basic idea is **garbage collection (memory compaction)**—reuse unused memory blocks.
- How to do garbage collection?
  - 1. Bounds register is used to store the highest (or lowest) location in memory accessible by each program.
  - 2. The relocation register contains the value, which could be positive or negative, and that must be added to each address referenced in the program so that the memory addresses are correctly accessed after relocation.

8. The drawback of relocatable dynamic partition

- **Overhead!**
- When should we use relocatable dynamic partition?
  - 1. When memory is busy.
  - 2. When many jobs are waiting.
  - 3. When the waiting time is too long.
- What can we learn from relocatable dynamic partition?
  - 1. Programs do not have to be stored completely in memory.
  - 2. Programs do not have to be stored contiguously in memory.