Memory Management
MFT Process Scheduling

• When process enters system, goes into a process queue
• Scheduler takes memory requirements of process, sizes of available regions
  • When a region of the right size becomes available, process moved into it
  • Process then goes on ready queue
  • When it finishes, memory region freed, new process brought in
MFT Memory Allocation

• Need to classify processes based on memory needs
  • User specifies maximum size
  • System can (try to) determine it automatically

• Methods take number of queues, region size, swapping, and scheduling algorithm into account
Methods: Multiple Queues

• Each memory region has its own associate queue, and process goes into queue of smallest region it will fit into

• Example:
  • System has 100K, 200K, and larger regions
  • 98K, 170K processes go into queues associated with 100K, 200K regions
Methods: Single Queues

• All processes go into 1 queue, and when scheduler selects next process to run, it waits for the partition to become available

• Example: 100K, 200K regions have their own queues
  • 98K process goes into queue associated with 100K region
  • 170K process goes into queue associated with 200K region
  • If 1MB partition became available, *neither* process would be put in it as they are not on its queue
Methods: Single Queue

• All processes go into 1 queue, and when scheduler selects next process to run, it waits for the partition to become available

• Example:
  • Each process has an associated region with it (use same as before)
  • 98K process enters queue, then 175K process enters queue
  • 200K region becomes free
  • As next process in ready queue goes into 100K region, not 200K region, it does not run until 100K region becomes free
  • Key point: 200K region remains empty until 100K region becomes free and 98K process moved into it
Methods: Single Queue

• All processes go into 1 queue, and scheduler goes down ready queue and picks next process that would fit into an associated region

• Example:
  • Each process has an associated region with it (use same as before)
  • 98K process enters queue, then 175K process enters queue
  • 200K region becomes free
  • Scheduler *skips* over 98K process (as that fits into 100K region)
  • Scheduler picks 175K process to run in 200K region

• Scheduler selects next process that fits into free partition *even of higher priority processes are ahead of it but are too large to run*
Methods: Single Queue

• All processes go into 1 queue, and scheduler goes down ready queue and picks next process that would fit into any free region

• Example:
  • 98K process enters queue, then 175K process enters queue
  • 200K region becomes free
  • Scheduler puts 98K process into 200K region as it is the first region that is free and that 98K job will fit
Methods: Single Queue + Swapping

• Swap processes based on which region they are in

• Example: 3 regions, and all jobs associated with a region scheduled using round robin
  • Quantum expires
  • Memory manager begins swapping out process in the region and swapping in another process associated with that region
  • CPU scheduler gives quantum to process in another region

• Memory manager must be able to swap processes fast enough so there are always processes in memory ready to execute when CPU is rescheduled
Methods: Single Queue + Swapping

• When high priority process comes in and a lower priority process is using the region where it would normally go, swap out lower priority process for the higher one

• When higher priority process done, swap lower priority process in
  • Called roll-out/roll-in

• Which region does a swapped process return to?
  • Static relocation: process must return to its original partition
  • Dynamic relocation: process can return to some other partition
Problems

• Process needs more memory than region has
  • MFT gives process fixed amount of memory

• How is this handled?
  • Terminate process
  • Return control to process with an error indication that request cannot be satisfied
  • Swap out process until a large enough region becomes available
    • Works only if using dynamic relocation
Problems

• Process needs more memory than region has
  • MFT gives process fixed amount of memory

• How is this handled?
  • Terminate process
  • Return control to process with an error indication that request cannot be satisfied
  • Swap out process until a large enough region becomes available
    • Works only if using dynamic relocation
Problems

• System has 100K available
• Almost all process are 20K, but one is 80K and only runs for ten minutes a day
• Then in best case, you can run 3 processes at a time, and you waste 60K (20K process in an 80K partition)
• So make the regions vary in size!
Preface to What Follows

- In MFT, address translation is static; that is, it is set when the program is loaded into memory
- Moving the program requires recalculating the relocation addresses
- Alternative: relocate addresses during execution
- This uses special hardware
Dynamic Relocation

• As each memory reference occurs, transform those that refer to main memory
  • As noted earlier, deals with sequence of memory references only

• Use base and limit (bounds) registers

• In hardware:
  • Check the memory reference does not exceed value in limit register; if it does, give error
  • Add contents of base register to memory reference
  • Access the transformed address
Address Translation

• Transformation of a virtual address into a physical one
• With this, address spaces can be moved during execution
  • And that’s dynamic relocation
• Note: limit register may contain physical address of end of address space
  • If so, reverse the order of checking and adding
  • Equivalent to earlier method
Hardware Requirements

- Base and limit registers
- Privileged instructions to store values in them
  - *Must* be privileged!
- Provided by a Memory Management Unit (MMU)
  - Now the MMU does a lot more; the basic idea is the above, though
- Processor status word (PSW) needs to indicate whether system is in privileged mode
  - Usually this is a set of bits
  - Sometimes PSW is called Processor Status Longword (PSL)
Hardware Requirements

• CPU must generate exceptions (aka traps, interrupts) when a process references memory outside its address space
  • Stop process execution
  • Jump to address indicated by interrupt/trap table
    • Each exception has address of routine to jump to
Operating System Requirements

• Operating system must track where in memory processes are, and what memory is not in use
  • Called a free list

• Allocate space to processes to be used as address space
  • Search the free list for chunk of memory of appropriate size

• Reclaim memory from terminated process
  • Put it onto free list so other processes can use it
Operating System Requirements

• Save and restore base and bounds registers during context switch
  • When execution switches from one process to another
  • Saved values put into area associated with single process
    • Usually Process Control Block (PCB), a collection of information about a process

• Handle exceptions
  • Functions to be called when exception occurs
  • Example: when bounds register exceeded, throw an exception, causing execution to transfer to function associated with attempt to access memory outside address space
  • Unless altered by process, exception handlers usually (but not always) terminate process
MVT

• Multiple Contiguous Variable Partition Allocation (MVT)
• Like MFT, but partition size varies dynamically
• Operating system tracks which parts of memory are in use
  • Free parts of memory often called *holes*
  • Done in a number of ways, such as bit maps, linked lists, skip lists, etc.
Example

- Processes placed in holes; if hole is too big, it is split and unused portion goes back onto free list
- At process termination, add its memory to free list
Memory Allocation Schemes

• **Best fit**: holes listed in order of increasing size
  - Process is put into the smallest hole it fits

• **Worst fit**: holes listed in order of decreasing size
  - Process is put into the first hole in the list

• **First fit**: holes listed in order of increasing base address
  - Process is put into the first hole it fits

• **Next fit**: like first-fit, except the search for a hole the job fits begins where the last one left off.

• (5) buddy system deals with memory in sizes of $2^i$ for $i < k$. There is a separate list for each size of hole. Put the job into a hole of the closest power of 2; if it takes up under half, return the unused half to the free list.
Memory Allocation Schemes

• **Buddy system**: Memory kept in sizes of $2^i$ for $i < k$
  - Separate list for each size of hole
  - Process put into hole of the closest power of 2
    - If it takes up under half, return unused half to the free list

• Example: memory of 16K, process requires 3K of memory
  - So needs to go into a 4K chunk

![Diagram of memory allocation scheme]

16K  →  8K  →  4K

3K goes here
Process Scheduling

• Scheduler keeps list of available block sizes, queue of processes waiting for memory
• Order jobs according to scheduling algorithm
• Allocate memory until not enough for next process
• Two approaches:
  • Skip to next process in queue that can fit into available memory
  • Wait until enough memory available for next process
Fragmentation

• Internal fragmentation: wasted space in partition
  • With MVT, little to no internal fragmentation

• External fragmentation: wasted space between partition
  • With MVT, much external fragmentation
Fragmentation

- Example: process 5 can run simultaneously with 1, 3, 4 were the two holes combined (56K); but they were not, so 56K of fragmentation
Compaction

• Moving contents of memory about in order to combine holes
• Example: in above, move 3’s memory in third figure to 1710K
  • Combines holes in 170K-200K and 230K-256K to get 1 hole in 200K-256K
  • Now 5 can run

• Need dynamic relocation
  • Copy contents of process memory
  • Update base register appropriately
Compaction Schemes

• Move all processes to one end of memory
  • Can get expensive in time

• Move enough processes to get needed amount of contiguous memory

• Example: CDC 6600 Scope Operating System kept 8 processes in main memory at a time
  • Used compaction on process termination
  • Kept 1 hole at bottom of main memory
Reducing External Fragmentation

• Reduce average process memory size
• Break memory in 2 parts, one for instructions, one for data
• Example: PDP-11 had 2 base/bounds register pairs
  • High order bit of each indicated which half of memory (high or low) the pair refers to
  • Instructions, read-only data go into high half of memory
  • Variables, etc. go into low half of memory
More on Memory Fragmentation

• Process needs $w$ words of memory
• Partition has $p$ words
• Internal fragmentation exists when $w - p > 0$
  • i.e., memory internal to partition not being used
• External fragmentation exists when $w - p < 0$
  • i.e., partition unused and available but is too small for any waiting process
Memory Fragmentation Example

• 22K of memory available
• Divided into partitions of sizes 4K, 4K, 4K, 10K
• In queue: 7K, 3K, 6K process memory requirements
  • 7K process goes into 10K partition; 3K internal fragmentation
  • 3K process goes into 4K partition; 1K internal fragmentation
  • 6K process waits
  • 2 4K partitions unused, so 8K external fragmentation
• Total fragmentation: 8K external, 4K internal, so 12K total
• Over 50% of memory in fragments!
Memory Fragmentation Example

• 22K of memory available

• Divided into partitions of sizes 4K, 8K, 10K

• In queue: 7K, 3K, 6K process memory requirements
  • 7K process goes into 8K partition; 1K internal fragmentation
  • 3K process goes into 4K partition; 1K internal fragmentation
  • 6K process goes into 10K partition; 4K internal fragmentation
  • all partitions used, so no external fragmentation

• Total fragmentation: 0K external, 6K internal, so 6K total

• Only 27% of memory in fragments