National University of Singapore

CS2106 Operating System

Second Half Summary Notes

Dong Shaocong A0148008J

April 17, 2018

1 Memory Management

Definition 1.1. Memory & OS responsibility

- **Memory** used to store: kernel code and data; user code and data
- **OS Responsibilities**:
  - Allocate memory to new processes.
  - Manage process memory.
  - Manage kernel memory for its own use.
  - Provide OS services to: get more memory, free memory, protect memory

Definition 1.2. Physical Memory Organization

- The actual matrix of capacitors (DRAM) or flip-flops (SRAM) that stores data and instructions.
- Arranged as an array of bytes.
- Memory addresses serve as byte indices.

Definition 1.3. Word: CPU data transfer unit

- 1 byte in 8-bit machines (ATMega328P, Intel 8080), 2 bytes in 16 bit machines (Intel 80286), 4 bytes in 32-bit machines (Intel Xeon), 8 bytes in 64-bit machines (Intel Celeron)

Definition 1.4. Endianness

How to store multibyte word (object > 1 byte), 2 general schemes, Eg: 0x87654321 (4 byte int)

- **Big Endian**: higher order bytes at lower addresses

	87	65	43	21
Address	a	a+1	a+2	a+3

- **Little Endian**: lower order bytes at lower addresses

	21	43	65	87
Address	a	a+1	a+2	a+3

x86: little endian ; Sparc: big endian ; powerpc: configurable

Definition 1.5. Alignment Issues

- **Data can be fetched across word boundaries.**

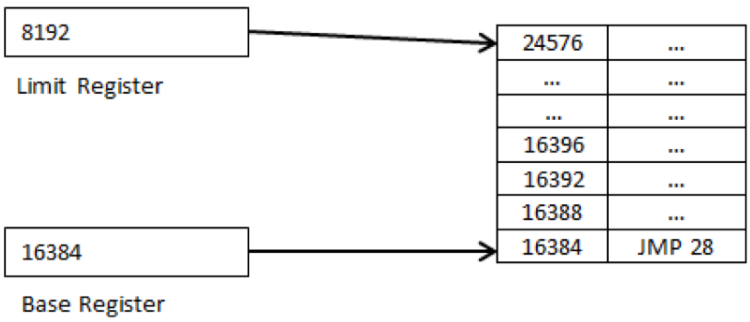
- E.g. fetching from address 1 in a 32-bit machine:
  - ✓Bytes from addresses 0 to 3 are fetched.
  - ✓Bytes from addresses 4 to 7 are fetched.
  - ✓Bytes from addresses 0, 5 to 7 are discarded.
  - ✓Bytes from addresses 1 to 3, 4 are merged.
- SLOW!!

Add unused bytes to ensure data structures always in units of words. Instructions fetched across word boundaries trigger "Bus Error" faults.

Why Memory Management?

- We want to use memory efficiently
- We want to protect processes from each other

- **Logical addresses**: These are the addresses as seen by executing processes code.
- **Physical addresses**: These are addresses that are actually sent to memory to retrieve data or instructions.



Note: **Having multiple processes complicates memory management**:

- **Conflicting addresses**: What if > 1 program expects to load at the same place in memory?
- **Access violations**: What if 1 program overwrites the code/data of another? Worse, what if 1 program overwrites parts of the operating system?
- The ideal situation would be to give each program a section of memory to work with. Basically each program will have its own address space!

To do this we require extra hardware support:

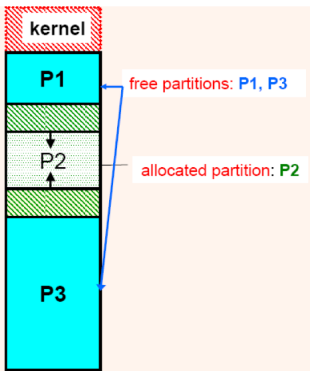
- **Base register**: This contains the starting address for the program. All program addresses are computed relative to this register.
- **Limit register**: This contains the length of the memory segment.

These registers solve both problems:

- We can resolve address conflicts by setting different values in the base register.
- If a program tries to access memory below the base register value or above the (base+limit) register value, a "segmentation fault" occurs!
- All memory references in the program are relative to the Base Register. E.g. "jmp 28" above will cause a jump to location 16412.
- Any memory access to location 24576 and above (or 16383 and below) will cause segmentation faults. (Other programs will occupy spaces above and below the segment given to the program shown here.)
- Base and limit registers allow us to partition memory for each running process: Each process has its own fixed partition, assuming that we know how much memory each process needs.

Definition 1.6. Fragmentation Issues

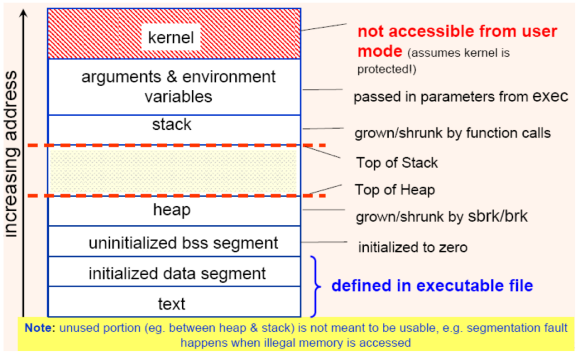
- **Internal fragmentation**:
  - Partition is much larger than is needed.
  - Cannot be used by other processes.
  - Extra space is wasted.
- **External fragmentation**:
  - Free memory is broken into small chunks by allocated memory.
  - Sufficient free memory in TOTAL, but individual chunks insufficient to fulfil requests.



Definition 1.7. Managing Memory within processes OS allocates memory for instructions. Global variables are created as part of the programs environment, and dont need to be specially managed. A **"stack"** is used to create local variables and store local addresses. A **"heap"** is used to create dynamic variables.

E.g. in UNIX, process space is divided into:

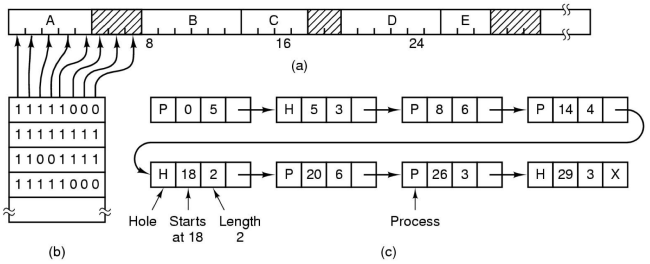
- **Text segments**: Read-only, contains code. May have ¿1 text segments.
- **Initialised Data**: Global data initialized from executable file. E.g. when you do:
  char *msg[]=“Hello world!”;
- **BSS Segment**: Contains uninitialized globals.
- **Stack**: Contains statically allocated local variables and arguments to functions, as well as return addresses.
- **Heap**: Contains dynamically allocated memory.



Definition 1.8. Managing Free Memory

Memory is divided up into fixed sized chunks called **“allocation units”**. Common sizes range from several bytes (e.g. 16 bytes) to several kilobytes,(a).

- **Bit maps**
- **Free/Allocated List**

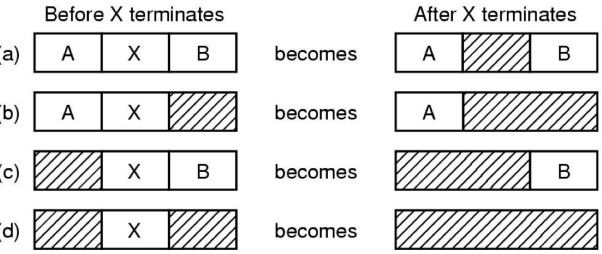


Bit Map (b)

- Each bit corresponds to an allocation unit. A “0” indicates a free unit, a “1” indicates an allocated unit.
- If a program requests for 128 bytes:
  1. Find how many allocation units are needed. E.g. if each unit is 16 bytes, this corresponds to 8 units.
  2. Scan through the list to find 8 consecutive 0’s.
  3. Allocate the memory found, and change the 0’s to 1’s.
- If a program frees 64 bytes: Mark the bits corresponding to the 4 allocation units as “0”.

Linked list (c)

- A single linked list is used to track allocated (“P”) units and free (“H”) units. Each node on the linked list also maintains where the block of free units start, and how many consecutive free units are present in that block.
- Allocating free memory becomes simple: Scan the list until we reach a “H” node that points to a block of a sufficient number of free units.
  - The list is implemented as a double-linked list.
    - Diagram below shows possible “neighbor combinations” that can occur when a process X terminates.
    - The “back pointer” in a double-linked list makes it easy to coalesce freed blocks together.

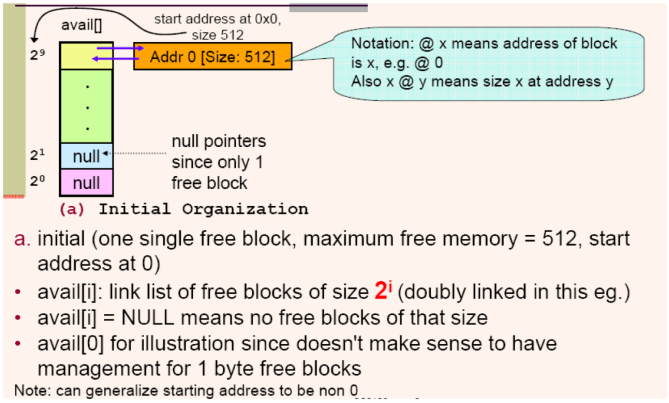


Definition 1.9. Allocation Policies

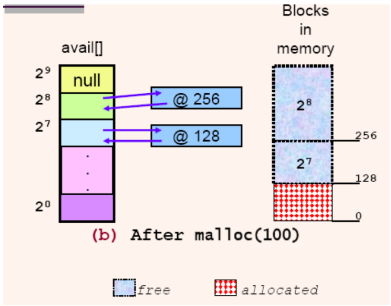
- **First Fit**:
  - Scan through the list/bit map and find the first block of free units that can fit the requested size.
  - Fast, easy to implement.
- **Best Fit**:
  - Scan through the list/bit map to find the smallest block of free units that can fit the requested size.
  - Theoretically should minimise "waste".
  - However can lead to scattered bits of tiny useless holes.
- **Worst Fit**:
  - Find the largest block of free memory.
  - Theoretically should reduce the number of tiny useless holes.
- **Note**: We can sort the free memory from smallest to largest for best fit (worst case unchanged), or largest to smallest for worst fit. This minimises search time. However coalescing free neighbours will be much harder.

Quick Fit: Buddy Allocation, Binary splitting

- Half of the block is allocated.
- The two halves are called "buddy blocks".
- Can coalesce again when two buddy blocks are free.



Example 1.1. Buddy Allocation



malloc(100)

1. Split 512 byte block into 2 blocks of 256 bytes.
2. Split one 256 byte block into two 128 byte blocks.

free(0): block Coalesces