

Memory management

- **Memory manager does:**
 - **keep track of memory usage**
 - **allocate memory**
 - **deallocate memory**
 - **swapping**
- **Issues in memory management**
 - **Transparency:**
 - * **want to let several processes coexist in main memory.**
 - * **no process should need to be aware of the fact that memory is shared.**
 - * **each must run regardless of the number and/or location of processes.**
 - **Safety:**

processes must not be able to corrupt each other.
 - **Efficiency:**

both CPU and memory should not be degraded badly by sharing.

- **Mono-programming:**

- **One user process + O.S. in memory.**
- **When user process is waiting for I/O, CPU idle.**

- **Multiprogramming:**

- **Several user processes in memory.**
- **One process waiting for I/O, another process can use CPU. Increase CPU utilization.**

- **Design issue in multiprogramming:**

- **Decide how many processes should be in memory to keep CPU busy.**

- **Probabilistic model**

- **n processes in memory**
- **Each process spends a fraction p of its time in I/O wait state**
- **Probability of all n processes are waiting for I/O: p^n**
- **CPU utilization: $1 - p^n$.**
- **n is called the degree of multiprogramming.**
- **Approximation of the model**

- **Example:**

Use this model to analyze the performance of a multiprogramming system.

Four jobs, $p = 80\%$.

- 1. Only job 1 in memory.**

20% of time CPU busy. 2 min CPU time finished within 10 min.

- 2. Jobs 1 and 2 in memory.**

36% of time CPU busy. Using round robin, each process uses CPU time $5 \times 0.36/2 = 0.9$ min within 5 min.

- 3. Jobs 1, 2 and 3 in memory.**

CPU busy = 0.49. Each process uses CPU $5 \times 0.49/3 \approx 0.8$ min.

⋮

Total time: 31.7 min for all four jobs.

If no multiprogramming, run four jobs one by one. The total time:

$$\frac{4+3+2+2}{0.2} = 55 \text{ min.}$$

Implementation of multiprogramming

- **Fixed partitions without swapping**
 - **Divide memory into n partitions (may be unequal)**

 - **Multiple input queues**

 - **Single input queue**

- **Relocation and protection**
 - **Relocation: Adjust a program to run in a different location of the memory**

 - **Protection:**
Prevent one job accesses the memory location of another job

– **Implementation**

- * **Relocation only:**
modify the instructions as the program is loaded into memory.

- * **Protection only (e.g. IBM 360):**
 - **Divide memory into 2K blocks**

 - **Assign 4-bit protection code to each block**

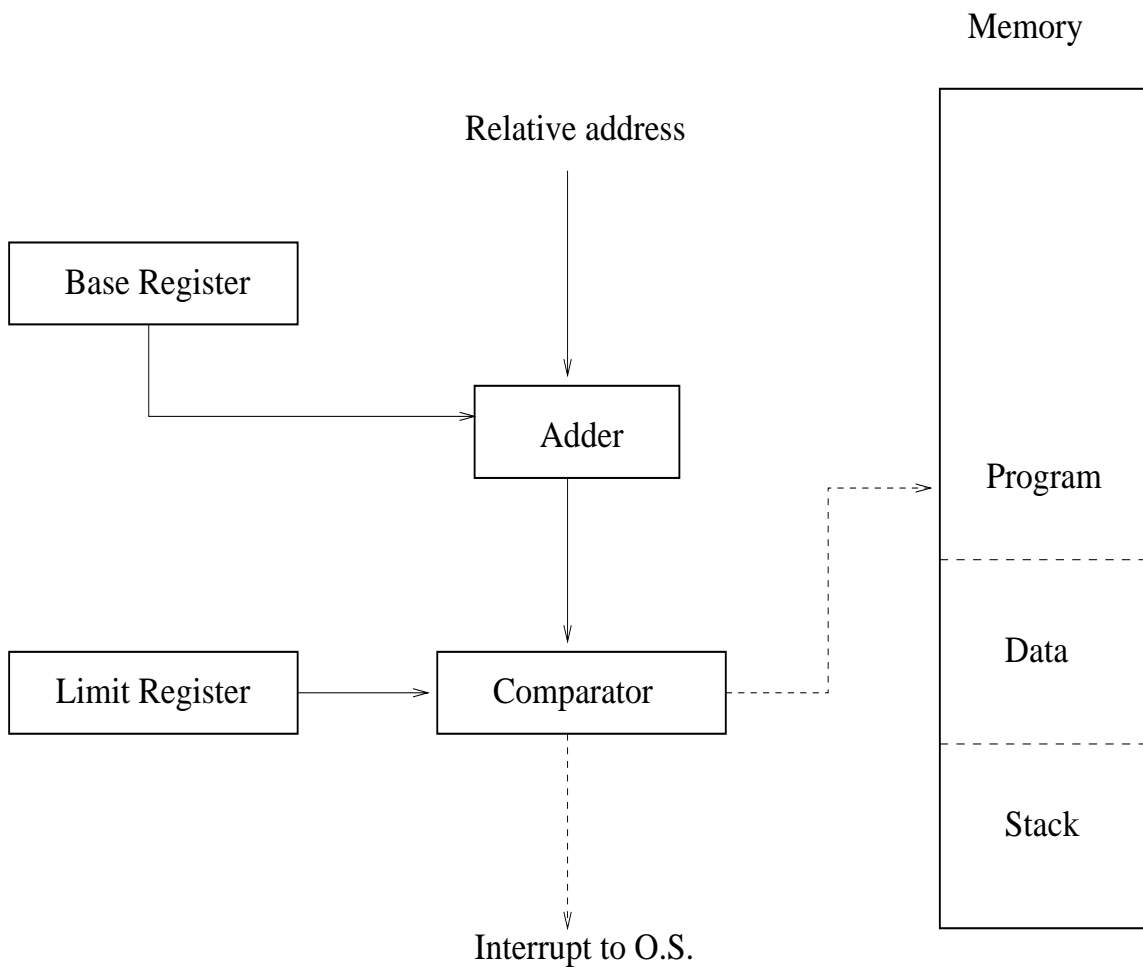
 - **PSW (Process State Word) contains a 4-bit key**

 - **When key \neq protection code, hardware trap**

 - **Only O.S. can change PSW key.**

*** Both relocation and protection**

- **Two special hardware registers:
base register and limit register.
Base register: stores the start address of
a partition
Limit register: stores the length of a par-
tition**
- **When a process is scheduled, O.S. sends
the start address of this process to base
register and the partition length to limit
register.**
- **Physical memory address =
(base) + memory addr. generated by
program**
- **O.S. checks the physical address against
the limit register**
- **After starting execution, still relocatable.**



Multiprogramming with variable partition

- **Difference from fixed partition: the number, location and size of the partitions vary dynamically.**

- **Concepts:**

Swapping: moving processes between memory and disk.

Memory compaction: merge all processes together to get a big hole.

- **How to deal with processes with growing data segment:**

Allocate a little extra memory.

Process grows too large: kill or swap out.

- **How to keep track of memory usage**

1. **Bit maps**

- **Divide memory into fixed size chunks (allocation units).**

Corresponding to each allocation unit is a bit in the bit map:

1: the unit is used

0: the unit is not used

- **Memory allocation:**

Find k consecutive 0 bits in the map for a process that needs k allocation unit memory.

- **Major drawback: slow**

2. Linked lists

– **Data structure:**

<i>P</i> or <i>H</i>	Start addr.	Length	Next entry
----------------------	-------------	--------	------------

P: process

H: hole

– **Sorted by address**

– **When a process terminates, four possibilities to merge**

– **Example**

– Allocation algorithms:

(a) First fit

Scan from the beginning and find the first hole that is big enough.

(b) Next fit

Similar to first fit except that search starts from where it left off.

(c) Best fit

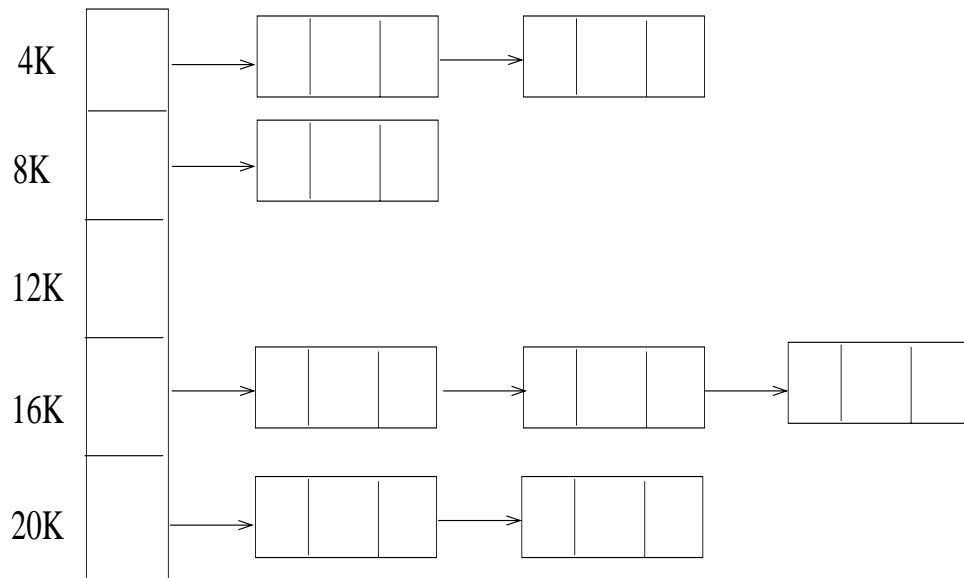
Search the entire list and takes the smallest hole that is adequate.

(d) Worst fit

Take the largest hole.

(e) Quick fit

Maintain separate lists of holes for commonly used sizes. Sorted by hole size.



Finding a hole is fast. What about merging holes?

(f) Buddy system

- * **Maintain a list of free blocks of size $2^0, 2^1, 2^2, \dots, 2^k, \dots$ bytes, up to the size of memory.**
- * **Example: 1 MB memory, 21 lists needed from 1 byte to 1 MB.**

Process A requests 70K, need 128K hole. 1MB is split into two 512K blocks (buddies).

- * **Allocation**
If the needed size is available, done.
If not, look at the next large size (double size). If available, split it into two blocks and use one.
If not available, look at next large size
.....
At most $\log N$ steps for N bytes memory.

* **Deallocation**

First search the same size queue to see if a merge is possible. If not, done.

Otherwise merge them and then search next large size queue until no merge possible.

At most $\log N$ steps.

* **Drawback:**

Low memory utilization.

Large internal fragmentation (wasted memory internal to the allocated segments).

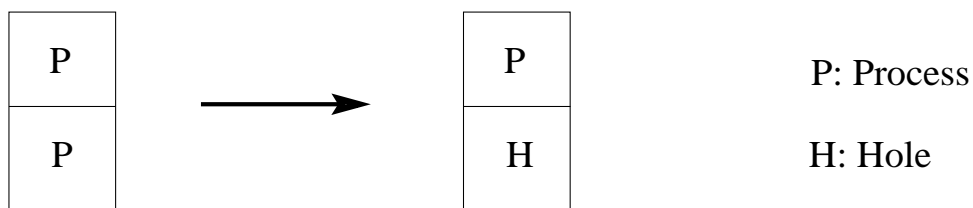
External fragmentation: wasted holes between the segments.

- **Analysis of swapping systems**

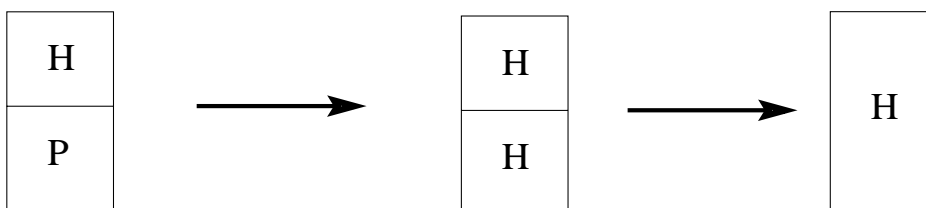
Estimate how much portion of memory is wasted as holes at any time.

Consider an average process.

After P finishes



50% of time



50% of time

Holes can be merged but processes cannot.

Fifty percent rule:

At steady state, if n processes then $\frac{n}{2}$ holes.

Unused memory rule: **f : fraction of memory occupied by holes** **s : average size of a process** **ks : average size of a hole** **m : total memory bytes**

$$\frac{n}{2}ks = m - ns$$

$$m = ns\left(1 + \frac{k}{2}\right)$$

$$f = \frac{\frac{n}{2} \cdot ks}{m} = \frac{\frac{k}{2} \cdot \frac{m}{1+k/2}}{m} = \frac{k}{k+2}$$

$$k = \frac{\mathbf{Hole\ size}}{\mathbf{Process\ size}}$$

Examples:

$$k = \frac{1}{2}: f = 20\%$$

$$k = \frac{1}{4}: f = 11\%$$

$$k = 2: f = 50\%$$

Virtual memory

- **Main idea: allow users to have a large logical address space without worrying the small physical memory.**
- **Goal: try to achieve access to disk is almost as fast as access to memory.**
- **How to do it: when size of program exceeds the size of physical memory, O.S. keeps part of the program currently in use in memory, and the rest on disk, with pieces of the program swapped between disk and memory as needed.**
- **Concepts:**
 - Virtual address: program generated address**
 - Physical address: the real memory address**
 - Without virtual memory:**
virtual address = physical address
- **Paging virtual memory (linear address space)**

- **Memory management unit (MMU):**
Maps logical address to physical address with a page table
- **Page size is fixed, and must be power of 2**
- **Example:**
16 bit virtual address (64K), 32K physical memory
Divide virtual address space into 4K pages
Divide physical address space into the same size page frames

- **Problems with paging**

- **Efficiency of access:**

- Even small page tables are generally too large to store in fast memory. So page table is kept in main memory.**

- One memory access requires two real memory accesses.**

- **Table space: smaller pages, larger page table**

- **Internal fragmentation: larger pages, more internal fragmentation.**

- **General page size: 512-8K**

- **Segmentation system**

- **idea: access files using memory address**
- **two dimension address space: segment and offset**

- **divide logical address space into segments. Each segment is a logical unit, such as data, code, and file.**

- **segment size is variable, and often large**

- **Segment table shows the mapping between logical addresses and physical addresses.**

- **Combine segmentation and paging system**
 - **Each segment must start at a page boundary. Divide each segment into pages and each segment has a page table.**

 - **Example:**
 - M68000-based microcomputer systems**
 - 16 processes**
 - Each process has 64 segments**
 - Each segment consists of 16 pages**
 - Each page has 4K bytes**

 - **Easy to share a file.**

- **Page fault: the page to be accessed is not in memory (a trap to CPU).**

- **What O.S. to do when a page fault occurs**
 - **Bring a page into memory**

 - **Update page table**

 - **Continue execution of process**

- **What to do when memory is full:
remove one page already in memory.**

- **Page replacement:
when a page fault occurs, O.S. removes a page from memory to make a room for the page that has to be brought in.**

**Why virtual memory can be successful:
because of locality of reference.**

The memory reference tends to be local. Access the nearby locations, not crossover the entire memory space.

Page replacement algorithms

- **Random algorithm**
Pick up any page at random.

- **Optimal algorithm**
Select for replacement that page for which the time to the next reference is the longest.
 - Results in the fewest number of page faults
 - Impossible to implement
 - Good for comparison

- **Not-recently-used algorithm (NRU)**

- **Based on locality of reference. Use the past to predict the future.**
- **Hardware support:**

Two bits associated with each page:

***R*: reference bit. Set by hardware on any memory read/write to the page.**

***M*: modified bit. Set by hardware when a page is written.**

O.S. can reset these two bits in software.

– **Algorithm:**

(1) When a process is started, R and M bits for all its pages are set to 0.

(2) On each clock tick interrupt, clear the R bit.

(3) When a page fault occurs, choose a page from the lowest numbered nonempty class:

Class	R bit	M bit	
0	0	0	not referenced, not modified
1	0	1	not referenced, modified
2	1	0	referenced, not modified
3	1	1	referenced, modified

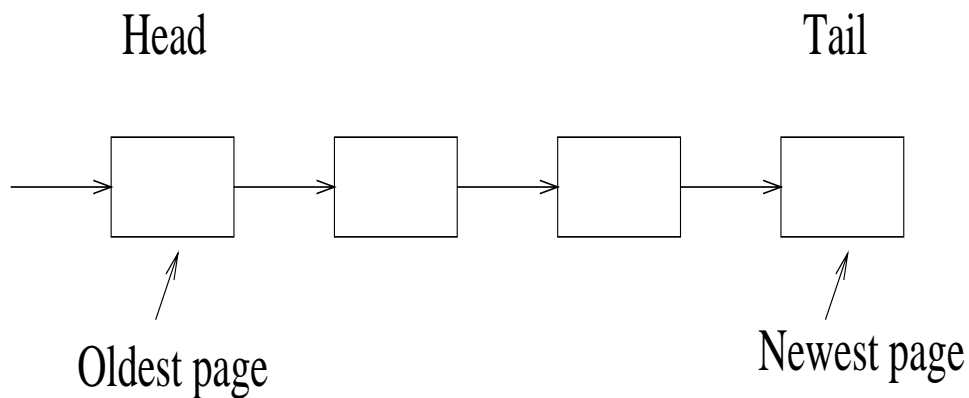
– **Easy to implement. Performance is ok.**

– **No history of how long the page is used.**

- **First-in-first-out (FIFO)**

Replace the page that has been in memory for the longest time.

- **O.S. maintains a list of all pages currently in memory:**



- **When page fault, remove the head page**
- **Problem: may remove a heavily used page**
- **Solution:**
Inspect R and M bits of all pages. Remove the oldest page with $R = 0$.

– **Two variants of FIFO**

* **Second chance**

Look at the oldest page. if $R = 0$, replace it. Otherwise, clear the R bit and put the page at the end of the list.

* **Clock**

Keep all the pages on a circular list.

– **An interesting thing in FIFO:**

More page frames may lead to more page faults.

- **Least-recently-used (LRU)**
Replace the page in memory that has not been referenced for the longest time.
 - **Performance close to optimal algorithm**
 - **Implementation is not cheap. Either requires special hardware or approximate software simulation.**
 - **Hardware implementation I (counter)**
 - * **A 64-bit counter, C**
 - * **Automatically incremented after each instruction**
 - * **After each memory reference, the current value of C is stored in page table for the page just referenced**
 - * **Lowest number in the page table is the least recently used**

– Hardware implementation II (matrix)

- * For n page frames, use an $n \times n$ bit matrix, initially all 0**

- * After page frame k is referenced, the hardware first sets row k to 1 and then set column k to 0**

- * The row whose binary value is lowest is the least recently used**

- **Software approximation I (not-frequently-used, NFU)**
 - * **Associate with each page a software counter, initially 0**
 - * **At each clock tick interrupt, O.S. adds each page's R bit to the counter associated with that page**
 - * **When a page fault occurs, replace the page with the lowest counter**
 - * **Problem: never forgets anything**

– **Software approximation II (aging algorithm)**

Modify NFU as follows:

- * **Counters are shifted right 1 bit before R bits is added in**
- * **R bit is added to the leftmost**
- * **Remove the lowest counter page**
- * **Different from LRU:**
 - **Cannot distinguish the references within the same clock tick**
 - **Counters have a finite number of bits, say, 8 bits, and can remember only the history of 8 clock ticks.**

- **Design issues for paging systems**

Aiming at good performance.

- **Working set:**
the set of pages a process is currently using
- **Thrashing:**
a program causes page fault every few instructions

How to control thrashing:

- * **Keep the entire working set in memory**
- * **Page fault frequency allocation algorithm (PFF)**
In most page replacement algorithms, we can choose a certain number of page frames for each process to keep a certain page fault rate. Dynamically allocate page frames to each process.

- **Optimum page size**

* **Page size p bytes**

Average internal fragmentation: $p/2$ bytes

Smaller page \Rightarrow smaller internal fragmentation

But smaller page \Rightarrow larger page table

* **Assume:**

n segments (logical units) in memory

Average process size s

Each page entry (in page table) requires e bytes

* **Memory wasted:**

$$\text{Overhead} = (s/p)e + p/2$$

* **Optimum page size**

$$p = \sqrt{2se}$$

* **Example:**

$s = 32\text{K}$ and $e = 8 \Rightarrow p = 724$ bytes.

Take 512 or 1024.

- **Not-recently-used (NRU) algorithm**
 - Need two hardware bits (R and M) per page
 - R is set at every reference and cleared after every clock tick
 - M is set at every write
 - Choose the smallest RM page to replace

- **First-in-first-out (FIFO) algorithm**
 - Replace the oldest page

- **Second chance algorithm**
 - “FIFO” + “R = 0”

- **Clock algorithm**
 - Similar to second chance, but difference implementation