

## Review

### CPU scheduling (Cont.)

- Scheduling algorithms
  - First in first out
  - Round robin scheduling
  - Priority scheduling
  - Priority classes
  - Shortest job first
    - \* Optimal with respect to average turnaround time.
    - \* General proof for  $n$  jobs.

## 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

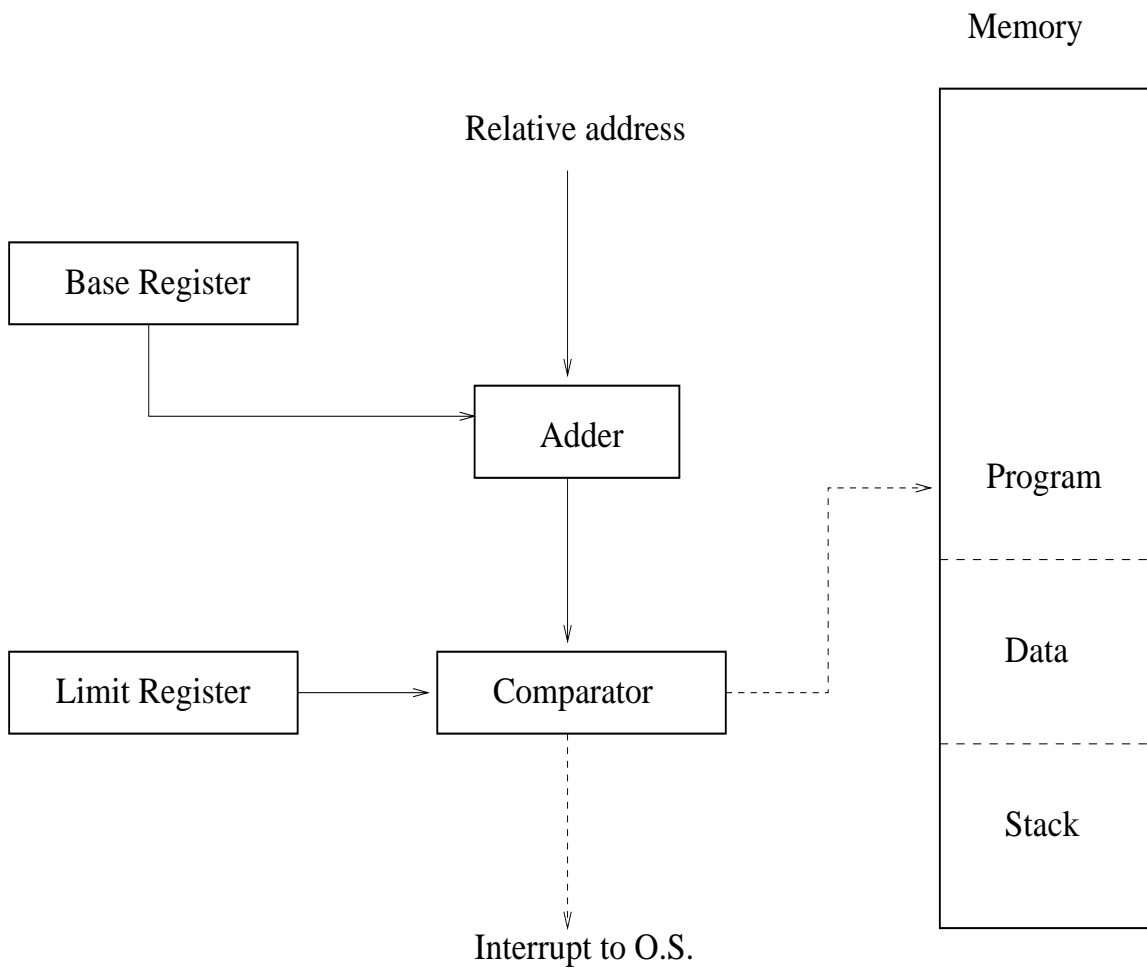
- Fixed partitions without swapping
  - Divide memory into  $n$  partitions (may be unequal)
  
  - Multiple input queues
  
  - Single input queue
  
- Relocation and protection
  - Relocation: Adjust a program to run in a different location of the memory
  
  - Protection:  
Prevent one job accesses 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 partition
- 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.



## Semaphore and shared memory system calls

### 1. Semaphore system calls

- Get a set of semaphores

- `semid = semget(key, nsems, permflags)`

**key:** a user defined name for the semaphore set

**nsems:** number of semaphores in the set

**permflags:** permission state (read, write)

**semid:** semaphore set identifier associated with key, used by other semaphore operations.

- Four ways to use it

(a) Create a private semaphore set

```
semid = semget(IPC_PRIVATE, nsems,  
0600|IPC_CREAT|IPC_EXCL)
```

**Return a unique semid system wide,  
private to the process.**

(b) Find key if already defined.

```
semid = semget(key, nsems, 0)
key ≠ IPC_PRIVATE, e.g. 0X200.
```

(c) Create only if key is not already defined

```
semid = semget(key, nsems,
0600|IPC_CREAT|IPC_EXCL)
If other processes specify the same
key, they will get the same semid.
```

(d) Find key if already defines, otherwise create

```
semid = semget(key, nsems, 0600|IPC_CREAT)
```

– SHELL commands for IPC:

`ipcs`: check ipc state

`ipcrm -s semid`: remove a semaphore.

- Semaphore control operations

`semval= semctl(semid, index, GETVAL, val)`

Get the value of the semaphore  
index: index in the set, e.g. 0 means the first semaphore in the set.

`semval= semctl(semid, index, SETVAL, val)`

Set the semaphore value to val.

`pid= semctl(semid, 0, GETPID, val)`

Return the process id of the last process that performs an operation on the semaphore.

- Semaphore operations (up and down)

`semop(semid, op_array, somevalue)`

`op_array`: an array of semaphore operations to perform

`somevalue`: the number of semaphore operation records

`struct sembuf op_array[somevalue]` has three fields:

`sem_num`: index to semaphore in the set

`sem_op`: -1: down; +1: up

`sem_flag`: usually set to `SEM_UNDO`, automatically “undo” all operations after process exits.

**Example:**

```
sem_num= 0;
```

```
sem_op = -1;
```

```
sem_flag = SEM_UNDO
```

## 2. Shared memory system calls

- Create shared memory segment

```
shmid = shmget(key, size,  
0600|IPC_CREAT|IPC_EXCL)
```

size: number of bytes.

- Shared memory operations

```
shmat(shmid, dataptr, flag)
```

Attach the memory segment identified by shmid to process's logical data space

dataptr = 0: the segment is attached to the first available address selected by the system

**dataptr nonzero:** attach to user specified address, depending on flag:

- flag & SHM\_RND is true. shmat will round dataptr to a page boundary
- flag & SHM\_RND is false. attach to the exact values of dataptr
- flag & SHM\_RDONLY is true. Read only.

**Example:**

```
struct databuf *pp;  
pp=(struct databuf *) shmat(shmid, 0,  
0);
```

- **Shared memory control operations**

`shmctl(shmid, command, &shm_stat)`

After you are done, remove the shared memory identifier specified by `shmid` from the system and destroy the shared memory segment and data structures associated to it:

`shmctl(shmid, IPC_RMID, (struct shm_id *)0)`

## Review

- Shortest Job Frist used in an interactive system (aging algorithm)
- Guaranteed scheduling
- Two-level scheduling

## Review

### Memory management

- Mono-programming
  
- Multiprogramming
  
- Probabilistic model for multiprogramming
  - CPU utilization:  $1 - p^n$ , where  $n$  is the degree of multiprogramming and  $p$  is the fraction of time a process waiting for I/O.
  
  - Analyze the performance of a multiprogramming system.
  
- Implementation of multiprogramming
  - Fixed partitions without swapping
  
  - Relocation and protection

## 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

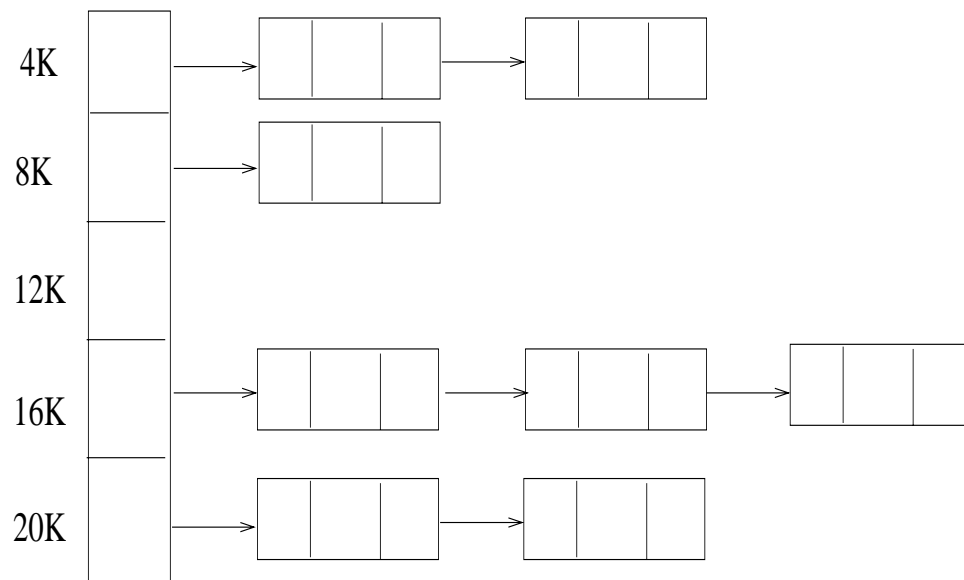
*H*: hole

- 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).

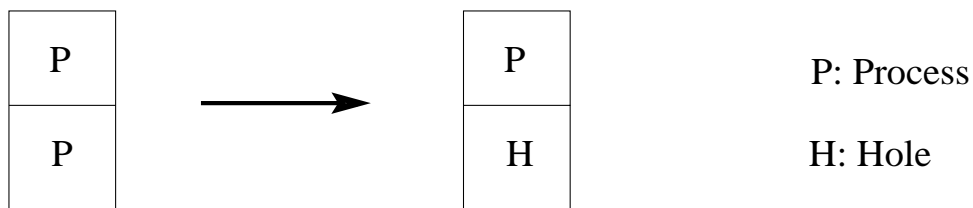
External fragmentation: wasted holes between the segments.

- Analysis of swapping systems

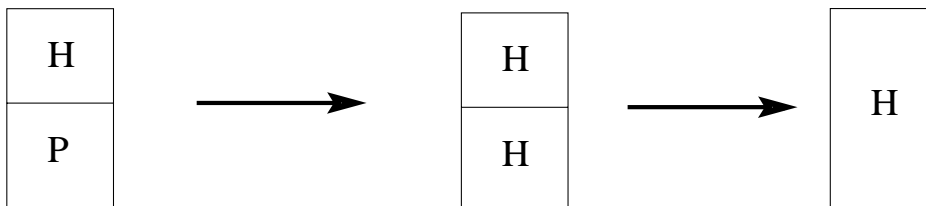
Estimate how much portion of memory is wasted as holes at any time.

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**

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

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

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

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

**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.