Operating System

Jin Zengrui

Computer System Overview

Basic ele: Processor, I/O Modules, Main Memory, System Bus

Processor: data processing

controls the operation of the computer

Central Processing Unit (CPU)

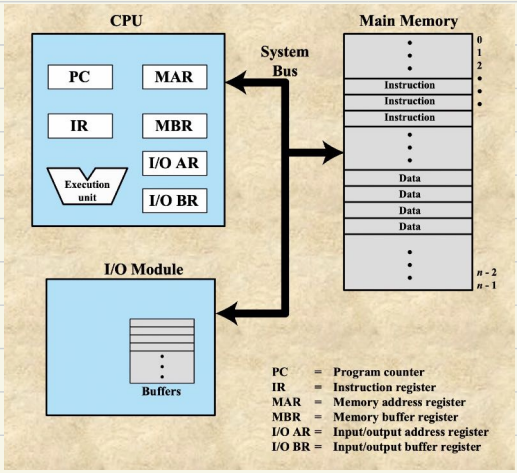
Main memory Volatile

Contents of the memory is lost when the computer is shut down

Referred to as real memory or primary memory

I/O Modules: moves data between the computer and external environments
e.g. storage, communications equipment, terminals

System Bus: communication among processors, main memory and I/O modules.



Instruction Fetch & Execute

processor fetches instruction from memory

program counter (PC) holds address of the instruction to be fetched next
PC is incremented after each fetch

Instruction Register (IR) : Fetched instruction is loaded into IR

Processor interprets the instruction and performs required action:
- [?] Processor-memory
- [?] Processor-I/O
- [?] Data processing
- [?] Control

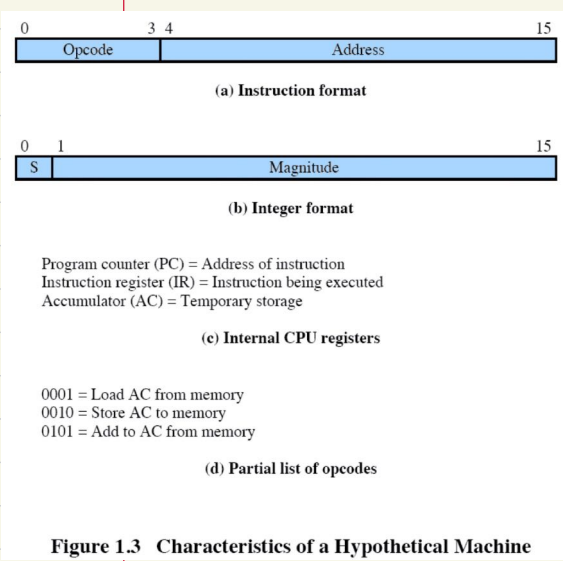


Figure 1.3 Characteristics of a Hypothetical Machine

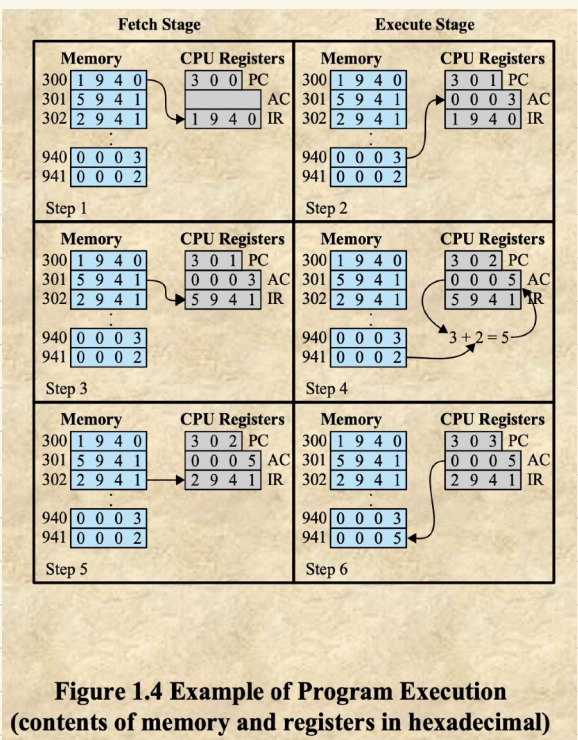
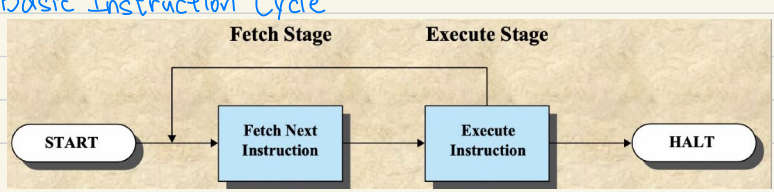


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

Basic Instruction Cycle



Interrupts

Interrupt the normal sequencing of the processor

To improve processor utilization
- most I/O devices are slower than the processor
- processor must pause to wait for device
- wasteful use of the processor

Classes of Interrupts

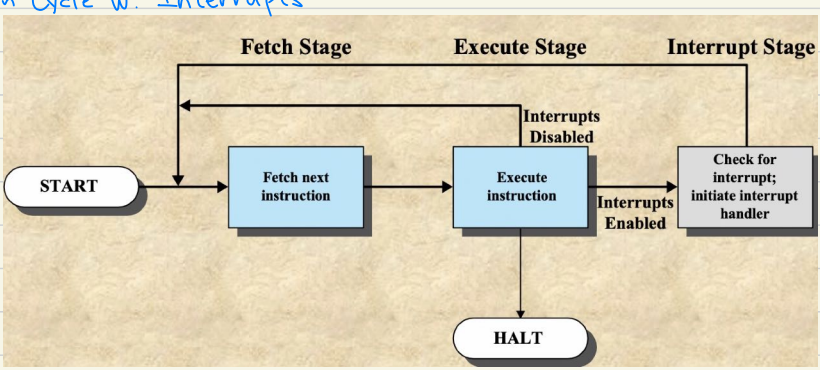
Program Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.

Timer Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.

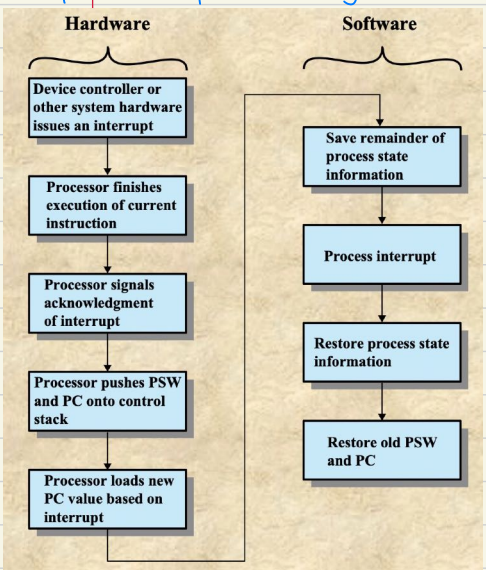
I/O Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.

Hardware failure Generated by a failure, such as power failure or memory parity error.

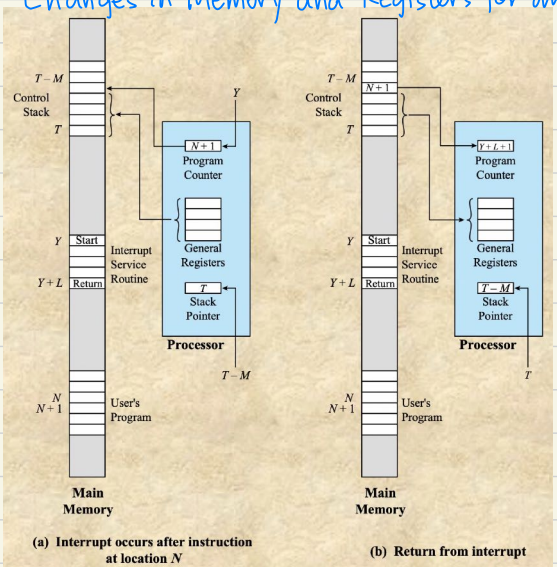
Instruction Cycle w. Interrupts



Simple Interrupt Processing



Changes in Memory and Registers for an Inter-rupt



Multiple Interrupts

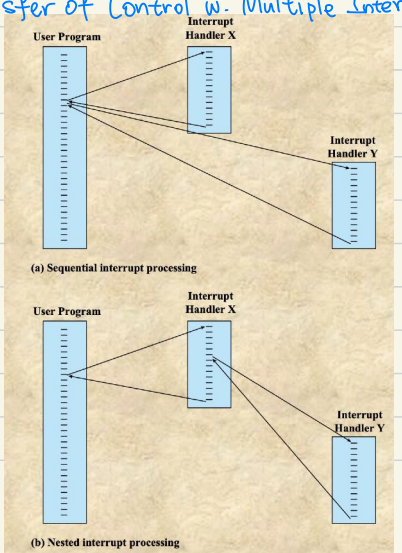
An interrupt occurs while another interrupt is being processed

Two approaches:

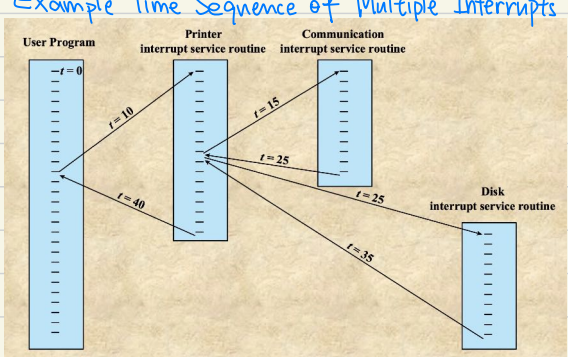
(1) disable interrupts while an interrupt is being processed

(2) use a priority scheme

Transfer of Control w. Multiple Interrupts



Example Time Sequence of Multiple Interrupts



Memory Hierarchy

Inboard Mem: Registers
Cache
Main Memory
Outboard Storage: Magnetic Disk
CD-ROM
CD-RW
DVD-RW
DVD-RAM
Blu-Ray
Off-line Storage: Magnetic Tape

Price ↓
capacity ↑
access time ↑
frequency of access to the memory by processor ↓

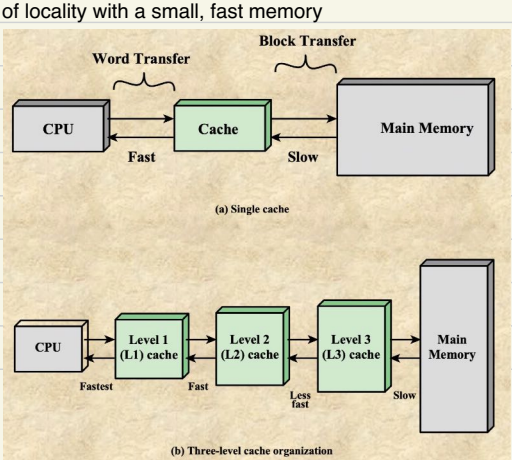
Principle of Locality:
Memory references by the processor tend to cluster

Can be applied across two levels of memory

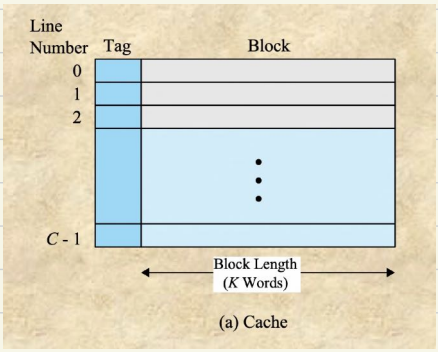
Secondary Memory (auxiliary memory)
- external
- nonvolatile
- used to store program and data files

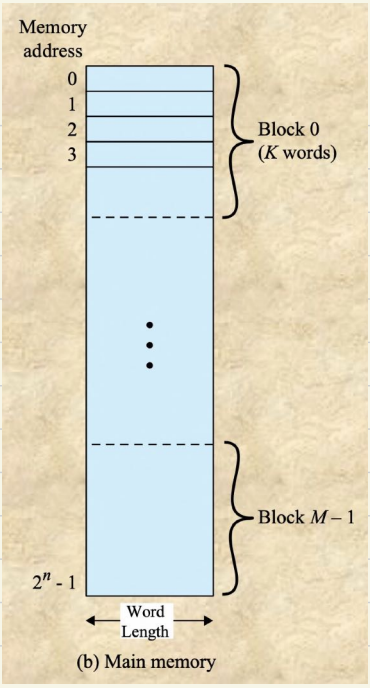
Cache Memory
- Invisible to the OS
- Interacts with other memory management hardware
- Processor must access memory at least once per instruction cycle
- Processor execution is limited by memory cycle time
- Exploit the principle of locality with a small, fast memory

Cache & Main Memory

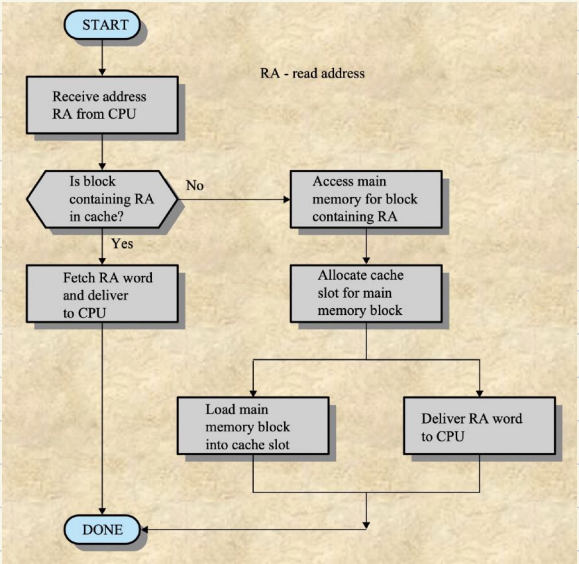


Cache / Main-Memory Structure





Cache Read Operation



Cache Design

Main categories: cache size: small caches have significant impact on performance
block size: the unit of data exchanged between cache & main mem
mapping function → determine which cache location the block will occupy
↳ constraints { when 1 block is read in, another may have to be replaced
the more flexible the mapping function, the more complex is the circuitry required to search the cache

★ replacement algorithm
Least Recently Used (LRU) Algorithm
effective strategy is to replace a block that has been in the cache the longest with no reference to it
hardware mechanisms are needed to identify the least recently use block — chooses which block to replace when a new block is to be loaded into the cache

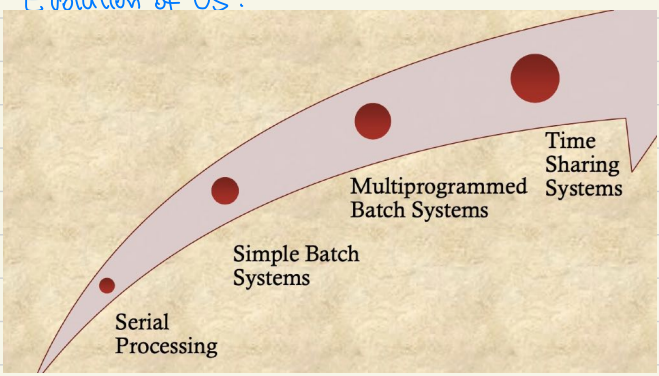
write policy
number of cache levels

Operating System Overview

Operating System Services

Program development
Program execution
Access I/O devices
Controlled access to files
System access
Error detection and response
Accounting

Evolution of OS:



Serial Processing:

No operating system
programmers interacted directly with the computer hardware
Computers ran from a console with display lights, toggle switches, some form of input device, and a printer
Users have access to the computer in "series"

Problems:

- Scheduling:
  - most installations used a hardcopy sign-up sheet to reserve computer time
  - time allocations could run short or long, resulting in wasted computer time
- Setup time
  - a considerable amount of time was spent just on setting up the program to run

Simple Batch Systems

- Early computers were very expensive
  - important to maximize processor utilization
- Monitor
  - user no longer has direct access to processor
  - job is submitted to computer operator who batches them together and places them on an input device
  - program branches back to the monitor when finished

Process Description & Control

Concurrency : Mutual Exclusion & Sync

Resource Competition: concurrent processes come into conflict when they are competing for use of the same resource

- need for mutual exclusion
- deadlock
- starvation

Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) May be initialized to a nonnegative integer value
2) The semWait operation decrements the value
3) The semSignal operation increments the value

Consequences:

There is no way to know before a process decrements a semaphore whether it will block or not
There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

Strong / Weak Semaphores

A queue is used to hold processes waiting on the semaphore

Strong semaphores

the process that has been blocked the longest is released from the queue first (FIFO)

Weak semaphores

the order in which processes are removed from the queue is not specified

Producer / Consumer Problems

General statement

one or more producers are generating data and placing these in a buffer
a single consumer is taking items out of the buffer one at a time
only one producer or consumer may access the buffer at any one time

The problem:

ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

Incorrect Solution (binary)

```
/* program producerconsumer */
int n;
binary semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Correct Solution (binary)

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Infinite Buffer

Non-binary semaphore

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Bounded-Buffer using Semaphores

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Concurrency : Deadlock and Starvation

Deadlock: The permanent blocking of a set of processes that either compete for system resources or communicate with each other

A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set

Permanent

No efficient solution

死锁定义为一组相互竞争系统资源或进行通信的进程间的“永久”阻塞。当一组进程中的每个进程都在等待某个事件（典型情况下是等待释放所请求的资源），而仅有这组进程中被阻塞的其他进程才可触发该事件时，就称这组进程发生了死锁。因为没有事件能够被触发，故死锁是永久性的。与并发进程管理中的其他问题不同，死锁问题并无有效的通用解决方案。

Resource Categories:

Reusable can be safely used by only one process at a time and is not depleted by that use
- processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable one that can be created (produced) and destroyed (consumed)
- interrupts, signals, messages, and information
- in I/O buffers

Example of 2 processes competing for Reusable Resources

Process P		Process Q	
Step	**Action**	**Step**	**Action**
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Consumable Resources Deadlock: Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

Deadlock occurs if the Receive is blocking

Conditions for Deadlock

Mutual Exclusion: only one process may use a resource at a time

Hold-and-Wait: a process may hold allocated resources while awaiting assignment of others

No Pre-emption: no resource can be forcibly removed from a process holding it

Circular Wait: a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

Dealing with Deadlock: 3 general approaches

Prevent Deadlock: adopt a policy that elimates one of the conditions

Avoid Deadlock: make the appropriate dynamic choices based on the current state of resource allocation

Detect Deadlock: attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention Strategy: 2 main methods

Indirect: prevent the occurance of one of the three necessary conditions

Direct: prevent the occurance of a circular wait

Deadlock Condition Prevention:

Mutual Exclusion: if access to a resource requires mutual exclusion then it must be supported by the OS

Hold and Wait: require that a process request all of its required resources at one time and blocking the process until all requests can be granted simutaneously.

No Pre-emption:

if a process holding certain resources is denied a further request, that process must release its original resources and request them again

OS may pre-empt the second process and require it to release its resources

Circular Wait

define a linear ordering of resource types

Deadlock Avoidance

A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock

Requires knowledge of future process requests

Two Approaches to Deadlock Avoidance

Resource Allocation Denial do not grant an incremental resource request to a process if this allocation might lead to deadlock

banker's algorithm

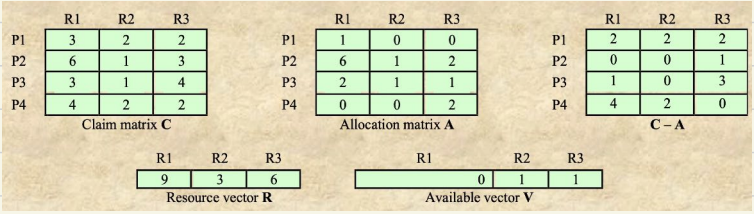
State of the system reflects the current allocation of resources to processes

Safe state is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock

Unsafe state is a state that is not safe

Example 1: Determination of a safe state

Initial State



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

P2 runs to completion

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

P3 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

Example 2: Determination of an unsafe state

Initial State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

P1 requests one unit each of R1 and R3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Process Initiation Denial do not start a process if its demands might lead to deadlock

Deadlock Avoidance Advantages

It is not necessary to preempt and rollback processes, as in deadlock detection
It is less restrictive than deadlock prevention

Deadlock Avoidance Restrictions

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Deadlock Strategies

Deadlock prevention strategies are very conservative

- limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- resource requests are granted whenever possible

Deadlock Detection Algorithms

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur

Advantages:

- it leads to early detection
- the algorithm is relatively simple

Disadvantage:

- frequent checks consume considerable processor time

Example: Deadlock Detection

Request matrix Q

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Allocation matrix A

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Resource vector

R1	R2	R3	R4	R5
2	1	1	2	1

Allocation vector

R1	R2	R3	R4	R5
0	0	0	0	1

Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

A Summary

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	`Conservative; undercommits resources`	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)
- No philosopher must starve to death (avoid deadlock and starvation)

1st Solution

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

2nd Solution

```
/*program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Memory Management

Terms

Frame — A fixed-length block of main memory.

Page — A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.

Segment — A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

Memory Management Requirements

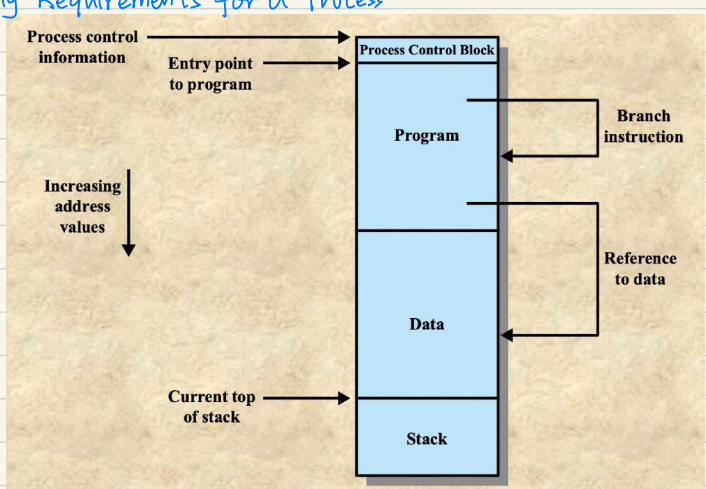
To satisfy the following requirements:

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
  - may need to *relocate* the process to a different area of memory

Addressing Requirements for a Process



Protection

Processes need to acquire permission to reference memory locations for reading or writing purposes
Location of a program in main memory is unpredictable
Memory references generated by a process must be checked at run time
Mechanisms that support relocation also support protection

Sharing

Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
Memory management must allow controlled access to shared areas of memory without compromising protection
Mechanisms used to support relocation support sharing capabilities

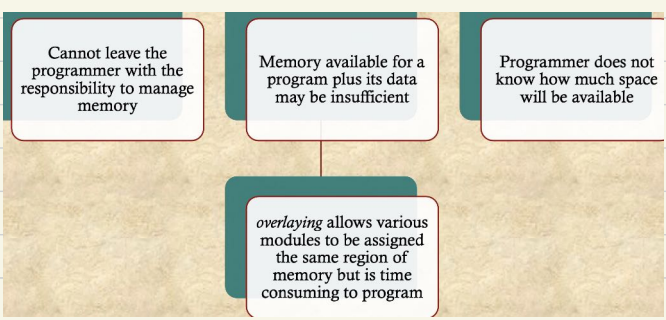
Logical Organization

Memory is organized as linear
Segmentation is the tool that most readily satisfies requirements

Programs are written in modules

- modules can be written and compiled independently
- different degrees of protection given to modules (read-only, execute-only)
- sharing on a module level corresponds to the user's way of viewing the problem

Physical Organization



Memory Partitioning

Memory management brings processes into main memory for execution by the processor
    involves virtual memory
    based on segmentation and paging
Partitioning
    used in several variations in some now-obsolete operating systems
    does not involve virtual memory

Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Example of Memory Partitioning

Equal-size partitions

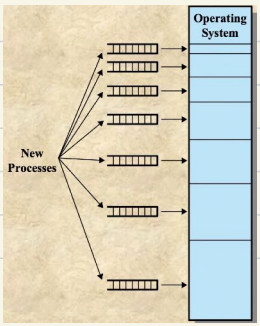
Unequal-size partitions

Disadvantages of Fixed Memory Partitioning

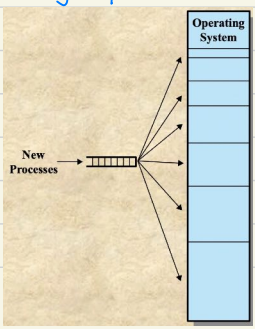
A program may be too big to fit in a partition
    program needs to be designed with the use of overlays
Main memory utilization is inefficient
    any program, regardless of size, occupies an entire partition
    internal fragmentation
        wasted space due to the block of data loaded being smaller than the partition

Example of Memory Assignment for Fixed Partitioning

One process queue per partition



single queue



Disadvantages of Fixed Memory Partitioning

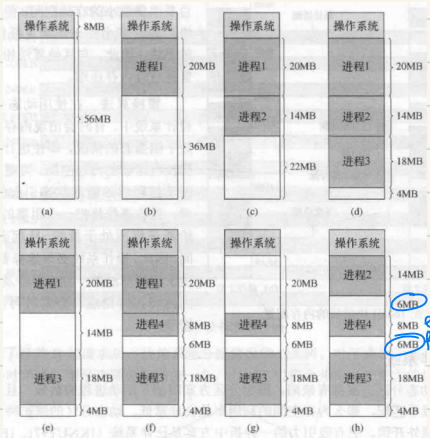
The number of partitions specified at system generation time limits the number of active processes in the system

Small jobs will not utilize partition space efficiently

Dynamic Partitioning

Partitions are of variable length and number

Process is allocated exactly as much memory as it requires



External Fragmentation 内存的碎片化

- memory becomes more and more fragmented
- memory utilization declines

Compaction

- technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- free memory is together in one block
- time consuming and wastes CPU time

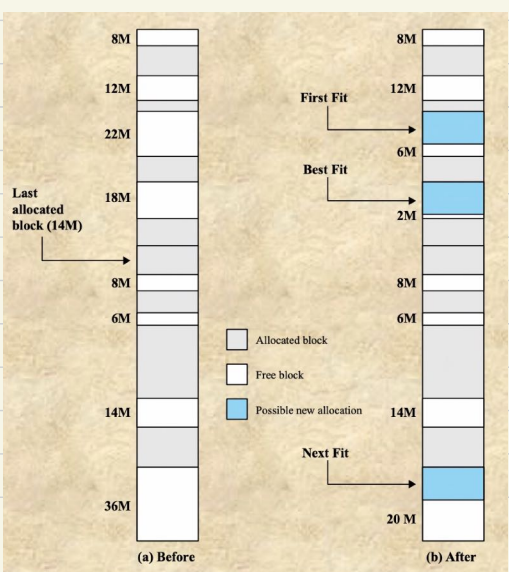
Placement Algorithms

Best-fit chooses the block that is closest in size to the request

First-fit begins to scan memory from the beginning and chooses the first available block that is large enough

Next-fit begins to scan memory from the location of the last placement and chooses the next available block that is large enough

Example Memory Configuration before and after Allocation of 16-Mbyte Block



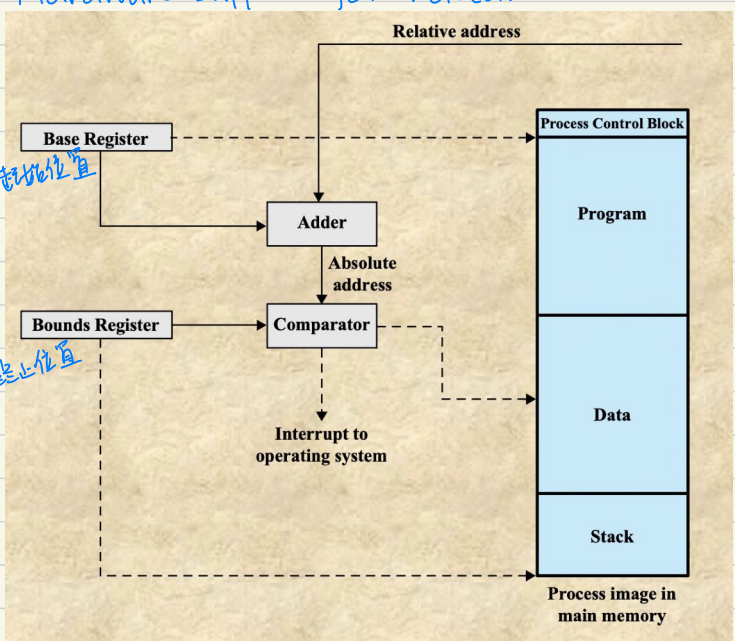
Addresses

Logical — reference to a memory location independent of the current assignment of data to memory

Relative — address is expressed as a location relative to some known point (通常是程序的开始处)

Physical or Absolute — actual location in main memory

Hardware support for relocation



程序在内存中的起始位置

程序在内存中的结束位置

protection 也是如此实现的

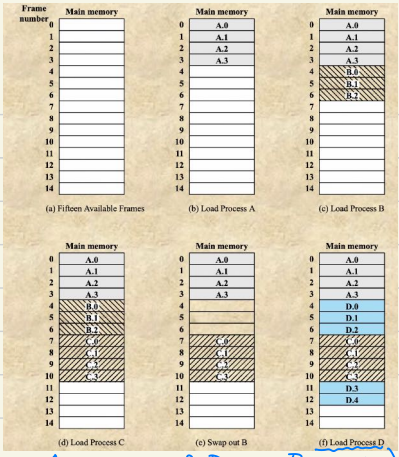
Paging

Partition memory into equal fixed-size chunks that are relatively small (frame)

内存和进程都划分成大小固定、相等的块

Process is also divided into small fixed-size chunks of the same size (page)

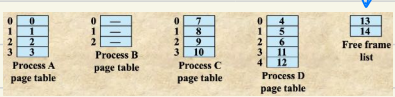
Pages: chunks of a process
Frames: available chunks of memory



Assignment of Process Pages to Free Frames

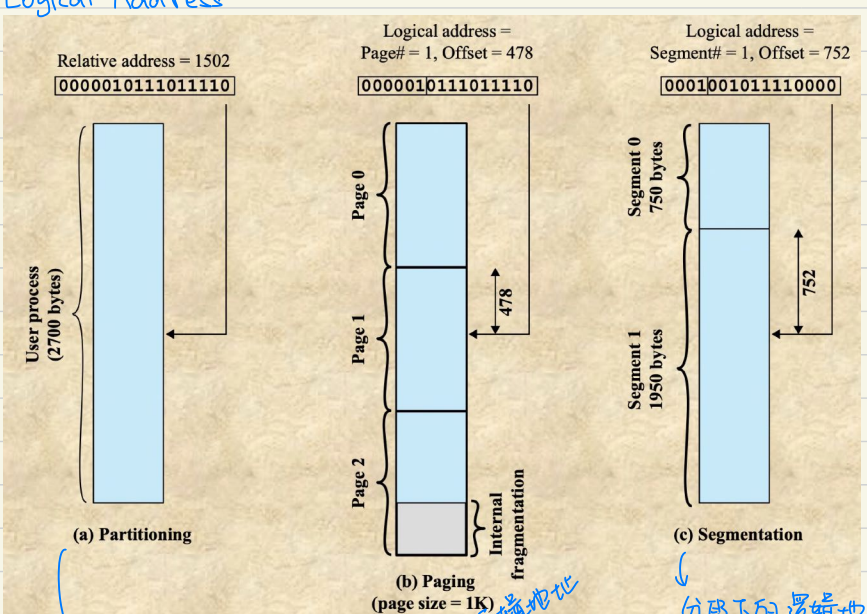
Page Table

Maintained by operating system for each process
Contains the frame location for each page in the process
Processor must know how to access for the current process
Used by processor to produce a physical address



Data structures at Time Epoch(f)

Logical Address

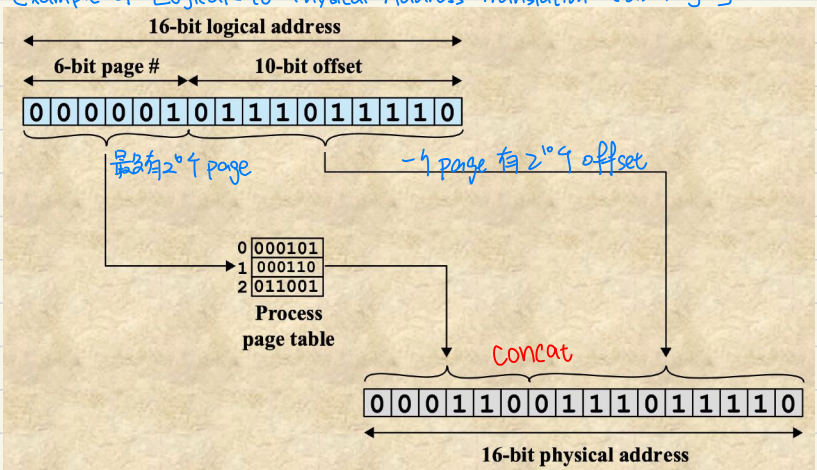


分段下的逻辑地址

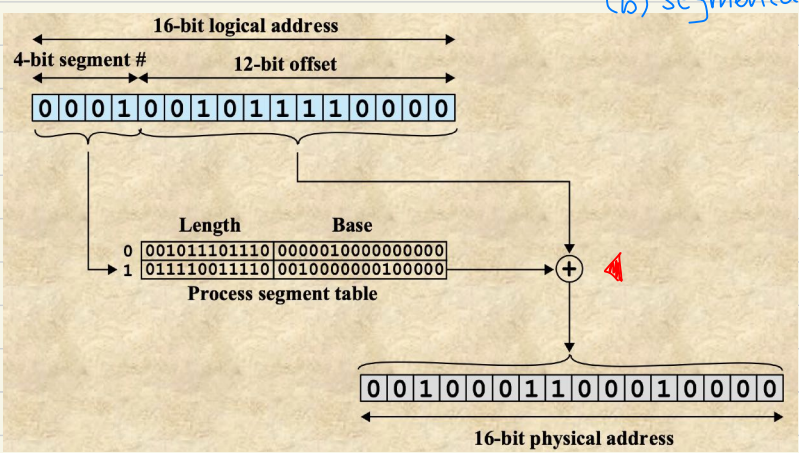
分页下的逻辑地址

分段下的逻辑地址

Example of Logical-to-Physical Address Translation: (a) Paging



(b) segmentation



Segmentation

A program can be divided into segments { may vary in length / there is a maximum length

Addressing consists of two parts { segment number / an offset

Similar to dynamic partitioning

Eliminates internal fragmentation

使用大小不等的 segment，所以分段类似于动态分区。

相同：不覆盖方案或 VM 的情况下，为执行一个程序，要将所有段都装入内存

不同：分段中，一个程序可以包括多个分区，且分区不要求为连续的

Segmentation Address Translation

Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses

The following steps are needed for address translation:

1. Extract the segment number as the leftmost n bits of the logical address
2. Use the segment number as an index into the process segment table to find the starting physical address of the segment
3. Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid
4. The desired physical address is the sum of the starting physical address of the segment plus the offset

与分页的 concatenation 不同！

Virtual Memory

Terms:

virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
virtual address space	The virtual storage assigned to a process.
address space	The range of memory addresses available to a process.
real address	The address of a storage location in main memory.

Hardware & Control Structures

Two characteristics fundamental to memory management:

(1) all memory references are logical addresses that are dynamically translated into physical addresses at run time

(2) a process may be broken up into a number of pieces that don't need to be contiguously located in main memory during execution

If these two characteristics are present, it is not necessary that all of the pages or segments of a process be in main memory during execution

Execution of a Process

- Operating system brings into main memory a few pieces of the program
- Resident set
  - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state
- Piece of process that contains the logical address is brought into main memory
  - operating system issues a disk I/O Read request
  - another process is dispatched to run while the disk I/O takes place
  - an interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state

Implications

- More processes may be maintained in main memory
  - only load in some of the pieces of each process
  - with so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory

Real and Virtual Memory

Real Memory : the actual RAM. main memory

Virtual Memory : (1) memory on disk

(2) allows for effective multiprogramming and relieves the user of tight constraints of main memory

Characteristics of Paging and Segmentation without VM

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames		Main memory not partitioned	
Program broken into pages by the compiler or memory management system		Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	
Internal fragmentation within frames		No internal fragmentation	
No external fragmentation		External fragmentation	
Operating system must maintain a page table for each process showing which frame each page occupies		Operating system must maintain a segment table for each process showing the load address and length of each segment	
Operating system must maintain a free frame list		Operating system must maintain a list of free holes in main memory	
Processor uses page number, offset to calculate absolute address		Processor uses segment number, offset to calculate absolute address	
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

Trashing A state in which the system spends most of its time swapping process pieces rather than executing instructions

solution

To avoid this, the OS tries to guess, based on recent history, which pieces are least likely to be used in the near future

Principle of Locality

Program and data references within a process tend to cluster
Only a few pieces of a process will be needed over a short period of time
Therefore it is possible to make intelligent guesses about which pieces will be needed in the future
Avoids thrashing

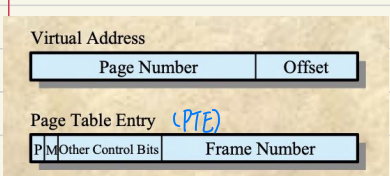
抖动 在请求分页存储管理中，从主存（DRAM）中刚刚换出（Swap Out）某一页面后（换出到Disk），根据请求马上又换入（Swap In）该页，这种反复换出换入的现象，称为系统颠簸，也叫系统抖动。产生该现象的主要原因是置换算法选择不当（即处理器大部分时间用于页的换出换入）

Paging

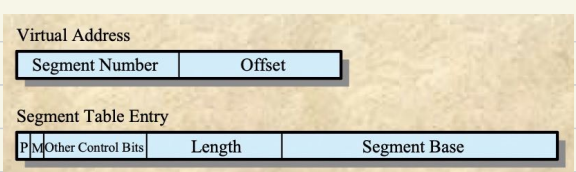
Virtual memory is usually associated with systems that employ paging

Each process has its own page table
each page table entry contains the frame number of the corresponding page in main memory

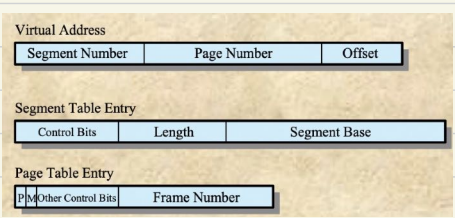
Typical Memory Management Formats



(a) Paging Only



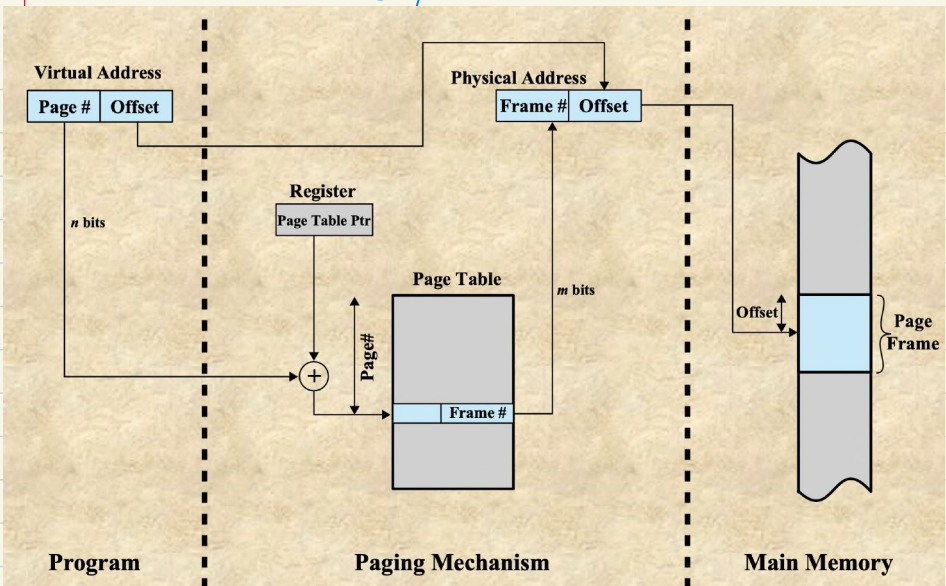
(b) Segmentation Only



(c) Combined segmentation and paging

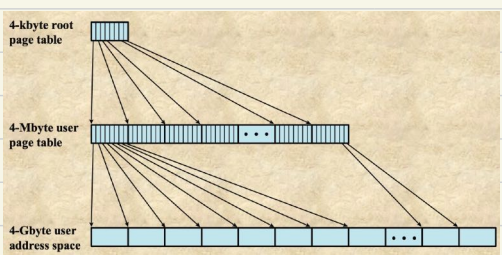
P = Present bit 对应的页是否在内存中
M = Modified bit 对应页的内容从上次装入内存到现在是否已改变

Address Translation in a Paging System

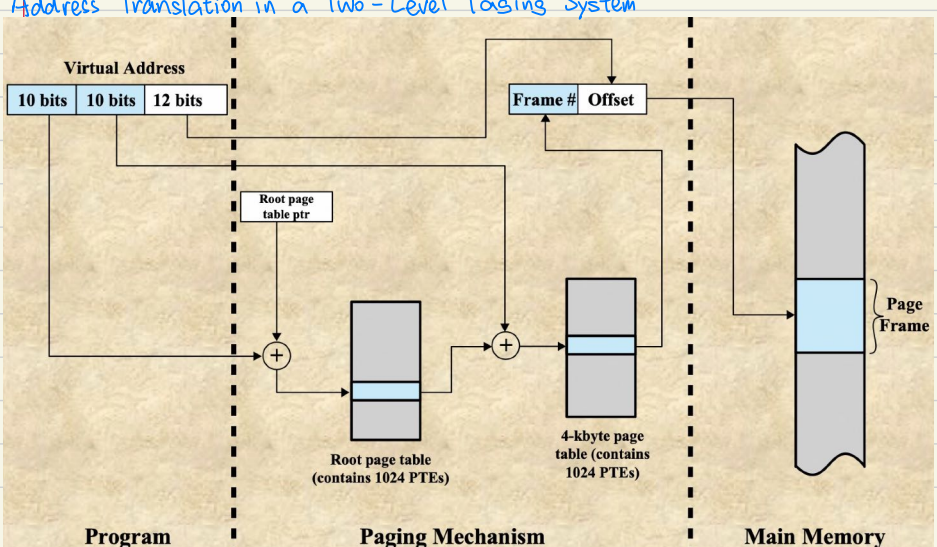


A Two-Level Hierarchical Page Table

避免页表占用内存过大的问题



Address Translation in a Two-Level Paging System



Inverted Page Table (解决上述页表大小与虚拟地址空间的大小成正比例问题)

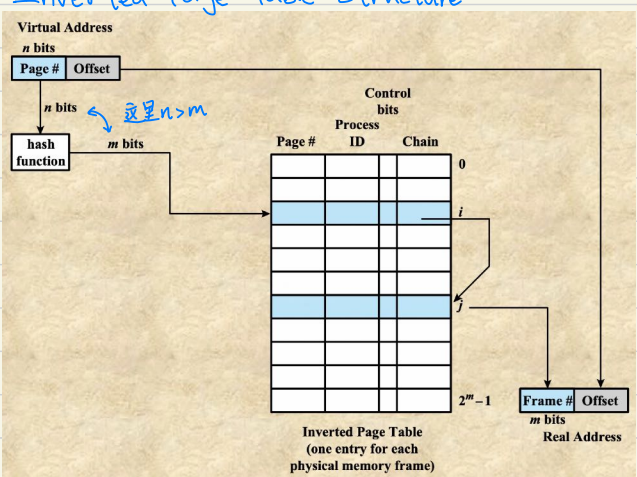
Page number portion of a virtual address is mapped into a hash value

hash value points to inverted page table

Fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported

Structure is called inverted because it indexes page table entries by frame number rather than by virtual page number

Inverted Page Table Structure



Each entry in Inverted Page Table includes:

- Page number
- Process identifier: the process that owns this page
- Control bits: includes flags and protection and locking information
- Chain pointer: the index value of the next entry in the chain

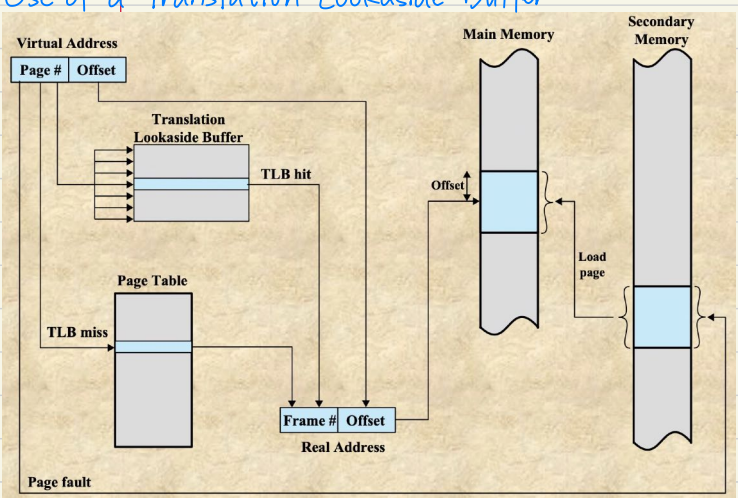
Translation Lookaside Buffer (TLB)

Each virtual memory reference can cause two physical memory accesses:

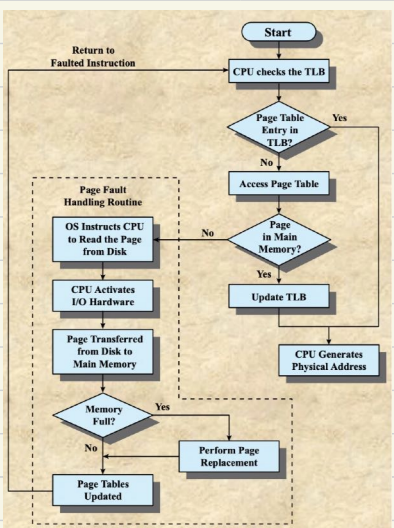
- one to fetch the page table entry
- one to fetch the data

To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a translation lookaside buffer

Use of a Translation Lookaside Buffer



Operation of Paging and Translation Lookaside Buffer (TLB)



Associative Mapping

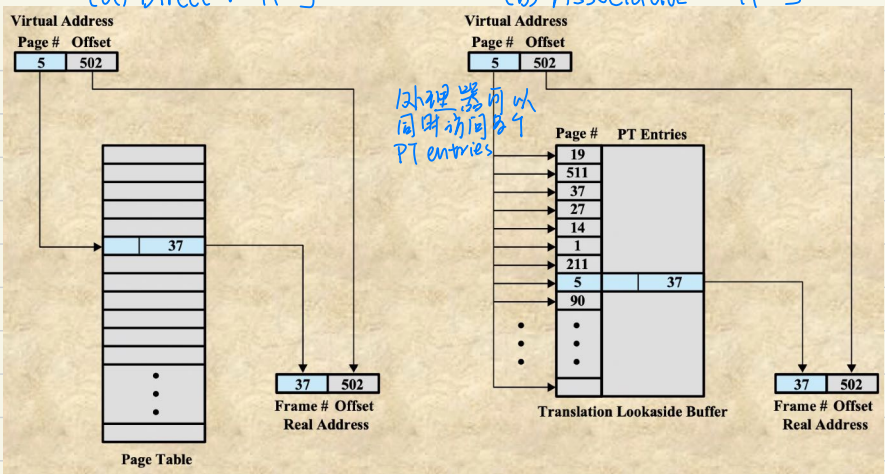
TLB only contains some of the page table entries so we cannot simply index into the TLB based on page number

- each TLB entry must include the page number as well as the complete page table entry

The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number

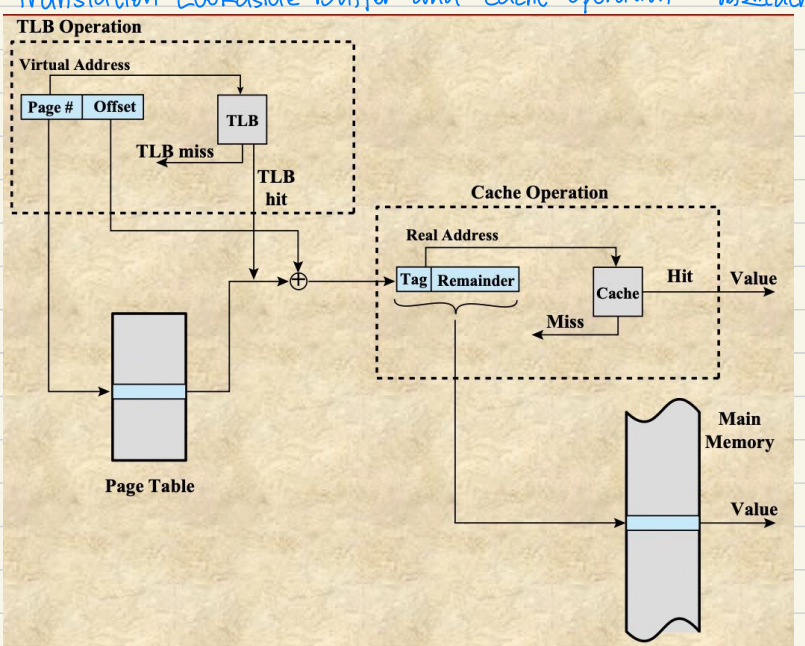
(a) Direct mapping

(b) Associative mapping



Direct Versus Associative Lookup for Page Table Entries

Translation Lookaside Buffer and Cache Operation



Page size

The smaller the page size, the lesser the amount of internal fragmentation

- however, more pages are required per process
- more pages per process means larger page tables
- for large programs in a heavily multiprogrammed environment some portion of the page tables of active processes must be in virtual memory instead of main memory
- the physical characteristics of most secondary-memory devices favor a larger page size for more efficient block transfer of data

The design issue of page size is related to the size of physical main memory and program size ⇨ main memory is getting larger and address space used by applications is also growing

⇩

most obvious on personal computers where applications are becoming increasingly complex

Contemporary programming techniques used in large programs tend to decrease the locality of references within a process

Segmentation

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments

Advantages:

- simplifies handling of growing data structures
- allows programs to be altered and recompiled independently
- lends itself to sharing data among processes
- lends itself to protection

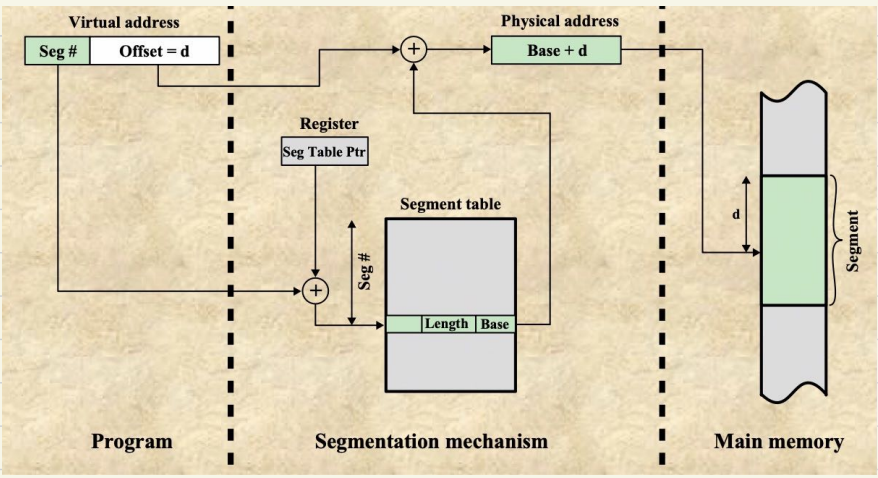
Segmentation Organization

Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment

A bit is needed to determine if the segment is in main memory.

Another bit is needed to determine if the segment has been modified since it was loaded in main memory

Address Translation in a Segmentation System



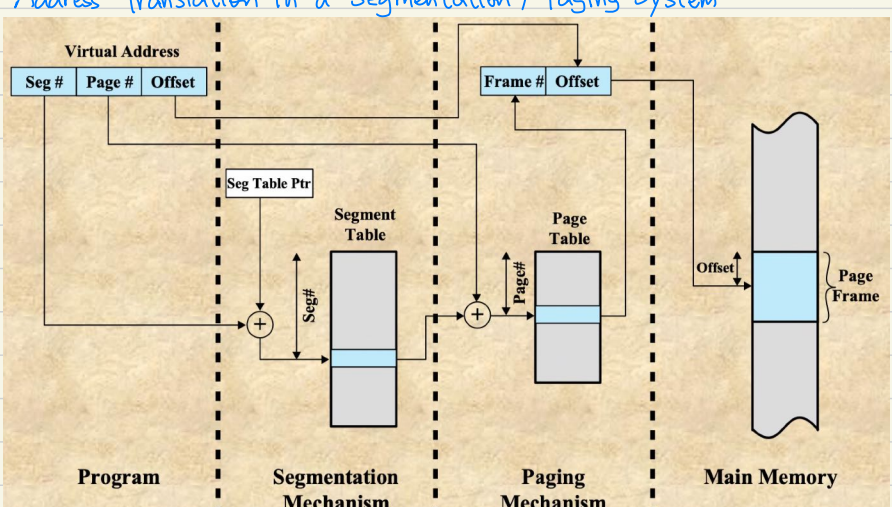
Combining Paging and Segmentation

In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

segmentation is visible to the programmer

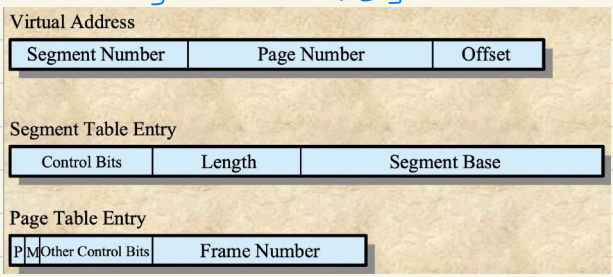
paging is transparent to the programm

Address Translation in a Segmentation / Paging System



用户地址分割为多个 Segment，每个 Segment 分割为多个 page，对应 Main memory 上的 frame

Combined segmentation and paging



P = Present bit 记得说好def
M = Modified bit

Operating System Software

Fetch policy

Determines when a page should be brought into memory

Two main types { Demand Paging
Prepaging

Demand Paging

1. only brings pages into main memory when a reference is made to a location on the page
2. many page faults when process is first started
3. principle of locality suggests that as more and more pages are brought in, most future references will be to pages that have recently been brought in, and page faults should drop to a very low level

Prepaging

1. pages other than the one demanded by a page fault are brought in
2. exploits the characteristics of most secondary memory devices
3. if pages of a process are stored contiguously in secondary memory it is more efficient to bring in a number of pages at one time
4. ineffective if extra pages are not referenced
5. should not be confused with “swapping”

Placement Policy

- Determines where in real memory a process piece is to reside
- Important design issue in a segmentation system
- Paging or combined paging with segmentation placing is irrelevant because hardware performs functions with equal efficiency

Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page must be brought in
  - objective is that the page that is removed be the page least likely to be referenced in the near future
- The more elaborate the replacement policy the greater the hardware and software overhead to implement it

Frame Locking

When a frame is locked the page currently stored in that frame may not be replaced

Kernel of the OS as well as key control structures are held in locked frames

I/O buffers and time-critical areas may be locked into main memory frames

locking is achieved by associating a lock bit with each frame

Basic Algorithms

Algorithms used for the selection of a page to replace

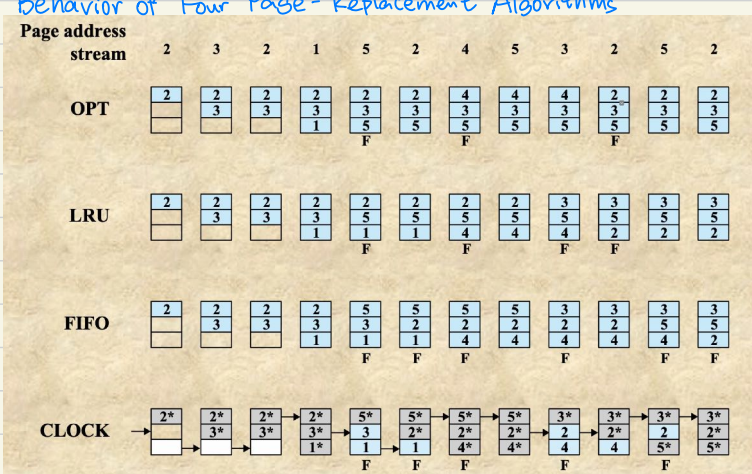
Optimal

Least recently used (LRU)

First-in-first-out (FIFO)

Clock

Behavior of Four Page-Replacement Algorithms



F = page fault occuring after the frame allocation is initially filled

page fault：需要用 page 不在 main memory 中，就是 page fault

OPT

要求选择置换下次访问距当前时间最长的页 不可能实现

Least Recently Used (LRU)

Replaces the page that has not been referenced for the longest time

By the principle of locality, this should be the page least likely to be referenced in the near future

Difficult to implement (难以实现，开销大)

One approach is to tag each page with the time of last reference

First-in-First-out (FIFO)

Treats page frames allocated to a process as a circular buffer

Pages are removed in round-robin style

simple replacement policy to implement (简单但性能较差)

Page that has been in memory the longest is replaced

Clock Policy

Requires the association of an additional bit with each frame

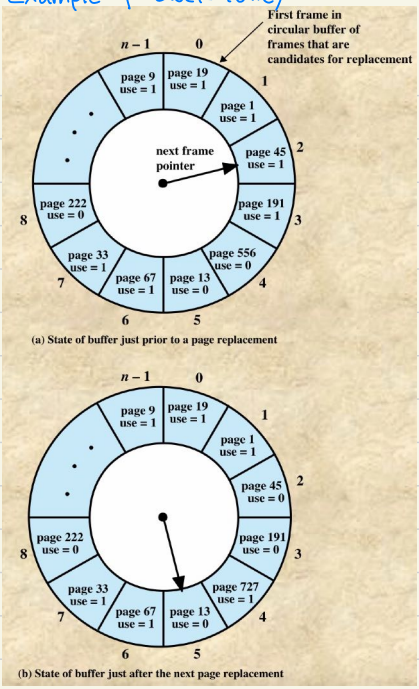
referred to as the use bit

When a page is first loaded in memory or referenced, the use bit is set to 1

The set of frames is considered to be a circular buffer

Any frame with a use bit of 1 is passed over by the algorithm

Example of Clock Policy



一页被置换后，指针指向buffer中的下一个frame

需要置换一页时，扫描 buffer，查找 use bit = 0 的一个 frame，每遇到一个 use bit = 1 的 frame，就将 use bit 置 0，然后跳到下一个 frame。

若所有 frame 的 use bit 均为 1，则全部置为 0，并回到起始位置，替换该 frame

也可配合 m 修改位来使用 Clock Policy

- 最近未被访问，也未被修改（$u = 0$；$m = 0$）
- 最近被访问，但未被修改（$u = 1$；$m = 0$）
- 最近未被访问，但被修改（$u = 0$；$m = 1$）
- 最近被访问，且被修改（$u = 1$；$m = 1$）

根据这一分类，时钟算法的执行过程如下：

1. 从指针的当前位置开始，扫描页框缓冲区。在这次扫描过程中，对使用位不做任何修改。选择遇到的第一个页框（$u = 0$；$m = 0$）用于置换。
2. 若第 1 步失败，则重新扫描，查找（$u = 0$；$m = 1$）的页框。选择第一个遇到的这种页框用于置换。在这一扫描过程中，将每个跳过的页框的使用位置为 0。
3. 若第 2 步失败，则指针回到其最初的位置，且集合中所有页框的使用位均为 0。重复第 1 步，并在必要时重复第 2 步。这样便可找到供置换的页框。

Page Buffering

Improves paging performance and allows the use of a simpler page replacement policy.

这些操作的一个重要特点是，被置换的页仍然留在内存中。因此，若进程访问该页，则可迅速返回该进程的驻留集，且代价很小。实际上，空闲页链表和修改页链表充当着页的高速缓存的角色。修改页链表还有另外一种很有用的功能：已修改的页按簇写回，而不是一次只写一页，因此大大减少了 I/O 操作的数量，进而减少了磁盘访问时间。

replaced pages are still in memory
but rather assigned to 2 lists.

- Free page list : list of page frames available for reading in pages
- Modified page list : pages are written out in clusters

Replacement Policy and Cache Size

Memory ↑ → Locality ↓ → Cache Size ↑

性能影响 ↓

1. if the page frame selected for replacement is in the cache, that cache block is lost as well as the page that it holds
2. in systems using page buffering, cache performance can be improved with a policy for page placement in the page buffer
3. most operating systems place pages by selecting an arbitrary page frame from the page buffer

Resident Set Management

resident set : the system must decide how many pages to bring into many memory when programs start up

tradeoff
1. the smaller the amount of memory allocated to each process, the more processes can reside in memory
2. small number of pages loaded increases page faults
3. beyond a certain size, further allocations of pages will not effect the page fault rate

due to locality

表 8.5 驻留集管理

	(局部置换)	(全局置换)
固定分配	• 分配给一个进程的页框数是固定的 • 从分配给该进程的页框中选择被置换的页	• 无此方案
可变分配	① • 分配给一个进程的页框数不时变化，用于保存该进程的工作集 ② • 从分配给该进程的页框中选择被置换的页	• 从内存中的所有可用页框中选择被置换的页；这将导致进程驻留集的大小不断变化
	③ Re-evaluate the allocation provided to the process and increase or decrease it to improve overall performance	

Replacement scope { local / global

(a) criteria used to determine resident set size
(b) timing of changes

Working Set of Process as Defined by Window Size

Working Set Strategy (改进FIFO，但与WS近似)

Sequence of Page References	Window Size, Δ = 2	Δ = 3	Δ = 4	Δ = 5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

用函数 $W(t, \Delta)$ 表示该进程在过去闭 Δ 个虚拟时间单位中被访问到的页集

虚拟时间定义：一系列内存访问 r(1), r(2), ..., r(i) 表示某个进程第 i 次产生的虚拟地址的页，时间通过内存访问来测量，即 t = 1, 2, 3 ... 表示进程内部虚拟时间

Δ↑ WT 即 $W(t, \Delta+1) \geq W(t, \Delta)$

该工作集同时还是一个关于时间的函数。若一个进程执行了 Δ 个时间单位，且仅使用一页，则有 $|W(t, \Delta)| = 1$。若许多不同的页可以快速定位，且窗口大小允许，则工作集可增长到和该进程的页数 N 一样大。因此有如下关系：

$$1 \leqslant W(t,\Delta) \leqslant \min(\Delta, N)$$

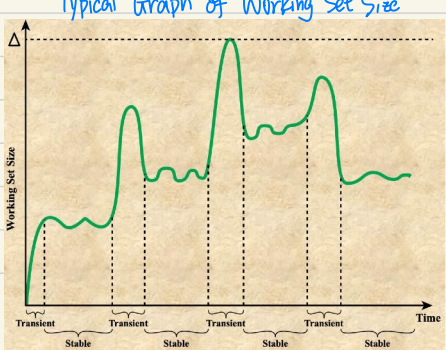
Transient

图 8.18 表明了对于固定的 Δ 值，工作集的大小随时间变化的一种方法。对于许多程序，工作集相对比较稳定的阶段和快速变化的阶段是交替出现的。当一个进程开始执行时，它访问新页的同时也逐渐建立起一个工作集。最终，根据局部性原理，该进程将相对稳定在由某些页构成的工作集上。接下来的瞬变阶段反映了该进程转移到一个新的局部性阶段。在瞬变阶段，来自原局部性阶段中的某些页仍然留在窗口 Δ 中，导致访问新页时工作集的大小剧增。当窗口滑过这些页访问后，工作集的大小减小，直到它仅包含那些满足新的局部性的页。

工作集的概念可用于指导有关驻留集大小的策略：依据局部性原理

1. 监视每个进程的工作集。
2. 周期性地从一个进程的驻留集中移去那些不在其工作集中的页。这基本上是一个 LRU 策略。
3. 只有当一个进程的工作集在内存中（即其驻留集包含了它的工作集）时，才可执行该进程

Typical Graph of Working Set Size



Working Set Strategy 的问题

1. 根据过去并不总能预测将来。工作集的大小和成员都会随时间而变化（例如，见图 8.18）。
2. 为每个进程真实地测量工作集是不实际的，它需要为每个进程的每次页访问使用该进程的虚拟时间作为时间标记，然后为每个进程维护一个基于时间顺序的页队列。
3. Δ 的最优值是未知的，且它在任何情况下都会变化。

"缺页率"：当高于某个阈值，即增加该系统的驻留集，使缺页中断减少.

另一种策略"缺页中断频率"
Page Fault Frequency (PFF)

每页有一个与之关联的使用位
该页被访问时使用位置为 1
发生缺页中断：记录从上次缺页中断到现在的虚拟时间（维护一个页访问计数器）
定义阈值 F，若从上一次缺页中断到这次的时间小于 F，则将该页加到该进程的 Resident Set 中
否则淘汰所有使用位为 0 的页，缩减 Resident Set，同时将其它页的使用位置 0。

缺页时间间隔与缺页率成反比

缺页中断发生的时间间隔和缺页率成反比。尽管最好是维持一个运行时的平均缺页率，但需要允许根据缺页率来决定驻留集的大小，使用时间间隔来度量是一种比较合理的折中。使用页缓冲对该策略进行补充时，会达到相当好的性能。

PFF 缺点之一：

PFF 方法的主要缺点之一是，如果要转移到新的局部性阶段，则在过渡过程中其执行效果不太好。对于 PFF，只有从上次访问开始经过 F 单位时间后还未再被访问的页，才会从驻留集中淘汰。而在局部性之间的过渡期间，快速而连续的缺页中断会导致该进程的驻留集在旧局部性中的页被逐出前快速膨胀。在内存突发请求高峰时，可能会产生不必要的进程去活和再激活，以及相应的切换和交换开销。

解决方案：Variable-Interval Sampled Working Set (VSWS)
Evaluates the working set of a process at sampling instances based on elapsed virtual time

Sampled Working Set，VSWS）策略[FERR83]。VSWS 策略根据经过的虚拟时间在采样实例中评估一个进程的工作集。在采样区间的开始处，该进程的所有驻留页的使用位被重置；在末尾处，只有在这个区间中被访问过的页才设置它们的使用位，这些页在下一个区间期间仍将保留在驻留集中，而其他页则被淘汰出驻留集。因此驻留集的大小只能在一个区间的末尾处减小。在每个区间中，任何缺页中断都将导致该页被添加到驻留集中；因此，在该区间中驻留集保持固定或增长。

VSWS 三个参数：
- M：采样区间的最大宽度
- L：采样区间的最小宽度
- Q：采样实例间允许发生的缺页中断数量

VSWS 策略如下：

1. 若从上次采样实例至今的单位时间达到 L，则挂起该进程并扫描使用位。
2. 若在这个长度为 L 的虚拟时间区间内，发生了 Q 次缺页中断：
   a. 若从上次采样实例至今的时间小于 M，则等待，直到经过的虚拟时间到达 M 时，才挂起该进程并扫描使用位。
   b. 若从上次采样实例至今的时间大于等于 M，则挂起该进程并扫描使用位。

选择参数值，使得上次扫描后发生第 Q 次缺页中断能正常地激活采样（情况 2b）。另两个参数（M 和 L）为异常条件提供边界保护。VSWS 策略试图通过增加采样频率，减少突然的局部性间过渡所引发的内存请求高峰，进而在缺页中断速度增加时，减少未使用页淘汰出驻留集的速度。这种技

Cleaning Policy

Concerned with determining when a modified page should be written out to secondary memory

demand cleaning: 当一页被选择置换时才被写回辅存

precleaning : 将已修改的多页在需要使用它们所占据的 frame 之前写回辅存 (in batch)

完全使用任何一种策略都存在危险。对于预约式清除，写回辅存的一页可能仍然留在内存中，直到页面置换算法指示它被移出。预约式清除允许成批地写回页，但这并无太大的意义，因为这些页中的大部分通常会在置换前又被修改。辅存的传送能力有限，因此不应浪费在实际上不太需要的清除操作上。

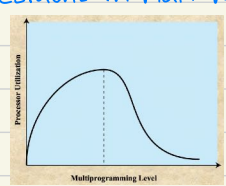
另一方面，对于请求式清除，写回已修改的一页和读入新页是成对出现的，且写回在读入之前。这种技术可以减少写页，但它意味着发生缺页中断的进程在解除阻塞之前必须等待两次页传送，而这可能会降低处理器的利用率。

一种较好的方法是结合页缓冲技术，这种技术允许采用下面的策略：只清除可用于置换的页，但去除了清除和置换操作间的成对关系。通过页缓冲，被置换页可放置在两个表中：修改表和未修改表。修改表中的页可以周期性地成批写出，并移到未修改表中。未修改表中的一页要么因为被访问到而被回收，要么在其页框分配给另一页时被淘汰。

Load Control

Determines the number of processes that will be resident in main memory

multiprogramming level



Critical in effective memory management

process ↓ many occasions when all processes will be blocked and much time will be spent in swapping

↑ lead to thrashing

Process Suspension

If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be swapped out

6种可能情况

Six possibilities exist:

- lowest-priority process
- faulting process
- last process activated
- process with the smallest resident set
- largest process
- process with the largest remaining execution window

- **最低优先级进程**：实现调度策略决策，与性能问题无关。
- **缺页中断进程**：原因在于很有可能是缺页中断任务的工作集还未驻留。因而挂起它对性能的影响最小。此外，由于它阻塞了一个一定会被阻塞的进程，并且消除了页面置换和 I/O 操作的开销，因而该选择可以立即收到成效。
- **最后一个被激活的进程**：这个进程的工作集最有可能还未驻留。
- **驻留集最小的进程**：在将来再次装入时的代价最小，但不利于局部性较小的程序。
- **最大空间的进程**：可在过量使用的内存中得到最多的空闲页框，使它不会很快又处于去活（deactivation）状态。
- **具有最大剩余执行窗口的进程**：在大多数进程调度方案中，一个进程在被中断或放置在就绪队列末尾之前，只运行一定的时间。这近似于最短处理时间优先的调度原则。

Uniprocessor Scheduling

Types of Scheduling
- Long-term scheduling
- Medium-term scheduling
- Short-term scheduling
- I/O scheduling

表 9.1 调度的类型

长程调度	决定加入待执行进程池
中程调度	决定加入部分或全部位于内存中的进程集合
短程调度	决定处理器执行哪个可运行进程
I/O 调度	决定可用 I/O 设备处理哪个进程挂起的 I/O 请求

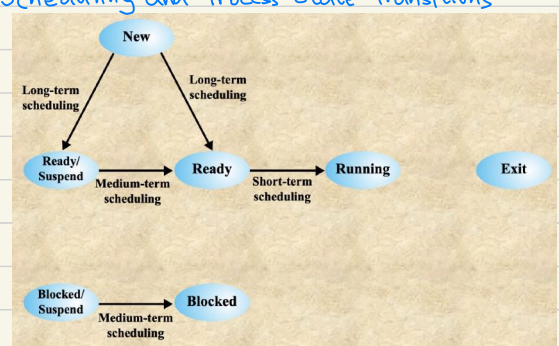
Processor Scheduling

Aim is to assign processes to be executed by the processor in a way that meets system objectives, such as response time, throughput, and processor efficiency

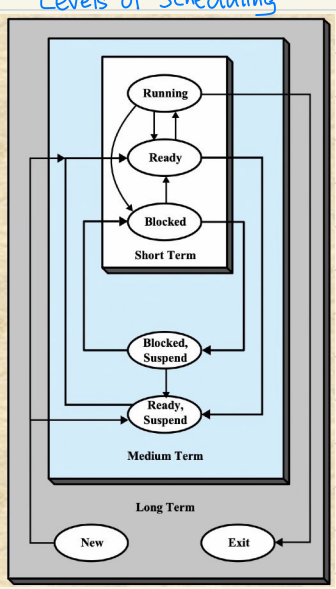
Three seperated function
- long-term scheduling
- medium-term scheduling
- short-term scheduling

} 由与系统并发有相关的性质驱动 multiprogramming level

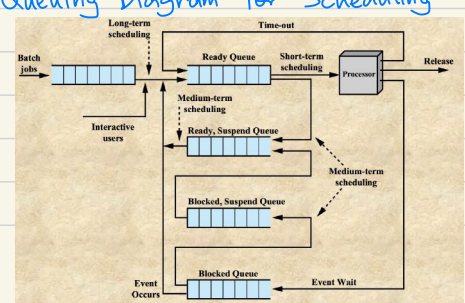
Scheduling and Process State Transitions



Levels of Scheduling



Queuing Diagram for Scheduling



本质上，调度属于队列管理问题
Queue Managing

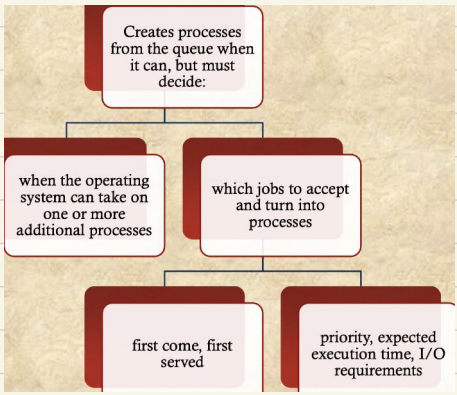
Long-Term Scheduler

Determines which programs are admitted to the system for processing

Controls the degree of multiprogramming

↑ processes created, ↓ the percentage of time that each process can be executed

may limit to provide satisfactory service to the current set of process



Medium-Term Scheduling

Part of the swapping function

Swapping-in decisions are based on the need to manage the degree of multiprogramming

considers the memory requirements of the swapped-out processes

Short-Term Scheduling

Known as dispatcher 分派程序

Executes most frequently

Makes the fine-grained decision of which process to execute next

Invoked when an event occurs that may lead to the blocking of the current process or that may provide an opportunity to pre-empt a currently running process in favor of another.

Examples:

- Clock interrupts
- I/O interrupts
- Operating system calls
- Signals (e.g., semaphores)

Short Term Scheduling Criteria

Main objective is to allocate processor time to optimize certain aspects of system behavior.

A set of criteria is needed to evaluate the scheduling policy

- **user-oriented criteria** (面向用户的)
  - relate to the behavior of criteria the system as perceived by the individual user or process (such as response time in an interactive system)
  - important on virtually all systems
- **system-oriented criteria** (面向系统的)
  - focus in on effective and efficient utilization of the processor (rate at which processes are completed)
  - generally of minor importance on single- user systems

Another criteria: Performance