Operating Systems. Memory Management

Contents I

Contents II

Contents III

Memory

- Hardware view: Electronic circuits to store and retrieve information.
- The *Bit* (**bi**nary elemen**t**) is the storage unit, the *byte* (8 bits) is the addressing unit.
- Although the byte is the address resolution unit, we'll consider *words*. The *Word* is the natural unit of data used by a particular processor design: the majority of the registers in a processor are usually word sized and the largest piece of data that can be transferred to and from the working memory in a single operation is a word.
    - Modern general purpose computers usually have a word size of 32 or 64 bits . . .
- For historical reasons it is frequent to say word = 2bytes = 16bits, double-word = 4bytes = 32 bits, quad-word = 8bytes = 64bits

Memory access

- According to the way used to access the information contained in the its cells, memory can be classified in :
  - **Conventional memory**: given a memory address (a number), the memory system returns the data stored at that address
  - **Associative memory** Content-addressable memories (CAM): given an input search data (tag), the CAM searches its entire memory to see if that tag is stored anywhere in it. If the tag found, the CAM returns a list of one or more storage addresses where the tag was found and it can also return the complete contents of that storage address. Thus, a CAM is the hardware embodiment of what in software terms would be called an associative array or hash table. Used in cache memories.

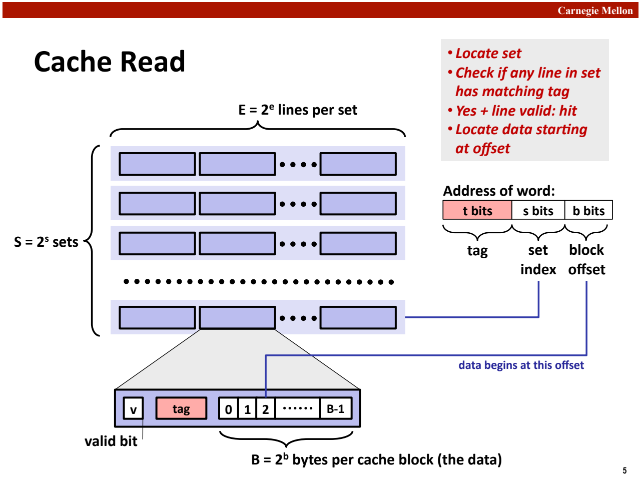
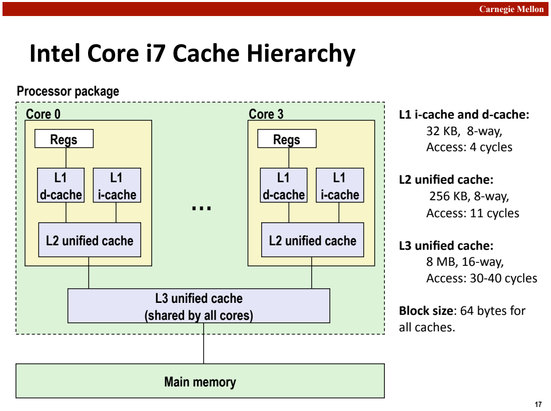


Figure: From R.E. Bryant et al. Computer Systems: A Programmer's Perspective (2nd edition), Pearson 2014



Intel Core i7 Cache Hierarchy

L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

Figure: From R.E. Bryant et al. Computer Systems: A Programmer's Perspective (2nd edition), Pearson 2014

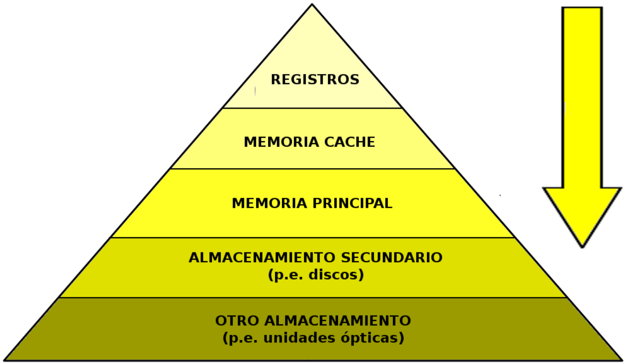
Memory hierarchy

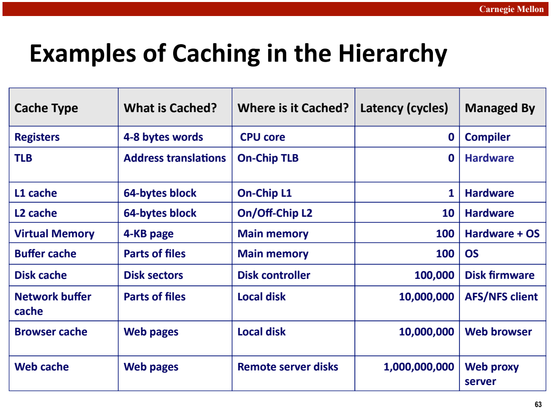
- The memory access time is the time required to access instructions or data in memory (read and write operations),
  - the time elapsed between the moment an address is set on the address bus and the moment the data is stored in memory or made available to the CPU
- It is desirable to have fast memories (short access times) with large storage capacity. Unfortulately the faster and the larger memory is, the higher its cost will be
- For this reason, faster and more expensive memories are used where memory accesses are more frequent.

Memory hierarchy

- These requirements led to the idea of Memory Hierarchy: memory is organised in layers according to access time and capacity
  1. Processor Registers
  2. Cache Memory
  3. Main Memory
  4. Hard Disk Drives
  5. Tape Drives and Optical Discs

Memory Hierarchy


REGISTROS
MEMORIA CACHE
MEMORIA PRINCIPAL
ALMACENAMIENTO SECUNDARIO (p.e. discos)
OTRO ALMACENAMIENTO (p.e. unidades ópticas)



Carnegie Mellon

Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

63

Figure: From R.E. Bryant et al. Computer Systems: A Programmer's Perspective (2nd edition), Pearson 2014

what is memory fragmentation

- File systems and memory can show internal and external fragmentation
    - **Internal fragmentation**: Wasted memory because assignation is made in blocks of n bytes and the requests of processes are not an exact multiple of n.
    - **External fragmentation**: Wasted memory that can not be assigned because it is not contiguous. External fragmentation appears in systems with (pure) segmentation..

- The Operating Systems is a resource manager, which implies:
  - The OS must keep the accounting of the resource memory (how much is free, how much is assigned to each process. . .)
  - The OS must have a policy for memory allocation
  - The OS must allocate memory to processes when they need it
  - The OS must recover memory allocated to processes when they no longer need it

Memory management

- The OS must keep the accounting of the system memory
  - The OS has to know the amount of free memory: otherwise, this memory could not be assigned to processes
  - The OS also has to register the memory allocated to each individual process (via zones in the process tables)
- Whenever a process is created or whenever a process requests memory, the OS allocates memory to that process
- When a process terminates the OS releases its allocated memory
- The OS also manages the virtual memory system

Segments for a process virtual address space

- A Process has diferent memory zones (sometimes called regions or segments) which differ in what it is stored in them
  - **Code (text).** The code for all the user (not kernel) functions in the process
  - **Static Data.** For external (global) and static C variables with initial values. Also for uninitialised external and static variables (BSS).
  - **Heap.** Dynamically allocated memory (via the *malloc* function in C). Usually the *static data* and the *heap* are contiguous with the *heap* in the upper part of what it is called *data*
  - **Stack.** Stack frames of function calls: arguments and local variables (automatic C vars), return addresses.

Memory management: example I

- Compile and run this C program

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

double t[1024*1024*32];  /*32 megas de doubles...unos 256 megas */
char * arrayPunteros[]={"hola","buenos","dias","que","lo","disfrutes",NULL,
NULL,NULL};


void regiones()
{
   static double pi =3.1415926 ;
   void *p = malloc(4096);

   printf ("variable local:     %p\n",&p);
   printf ("static inicializada:%p\n",&pi);
   printf ("global inicializada:%p\n",arrayPunteros);
   printf ("global sin inicial.:%p\n",t);
   printf ("constante:          %p\n",arrayPunteros[0]);
```

Memory management: example II

```
   printf ("puntero:             %p\n",p);
   printf ("brk:                 %p\n",sbrk(0));
   printf ("funcion:             %p\n",regiones);
   printf ("funcion libreria     %p\n",printf);
}

int main(int argc, char *argv[])
{
  int tiempo=20;
  pid_t pid=getpid();

  regiones();
  printf ("mi pid es %lu\n",(unsigned long) pid);
  printf ("hay unos %d segs para hacer en otro terminal 'pmap %lu'
             y ver las zonas\n",tiempo, (unsigned long) pid);
  sleep (tiempo);
  exit(0);
}
```

Memory management: example

- Output (linux 64 bit system)

```
$ ./a.out
variable local:        0x7fff2f1416a8
static inicializada:0x5641d8cf2050
global inicializada:0x5641d8cf2060
global sin inicial.:0x5641d8cf20e0
constante:             0x5641d8cf0008
puntero:               0x5641ea5852a0
brk:                   0x5641ea5a6000
funcion:               0x5641d8cef175
funcion libreria       0x7f02b5affcf0
mi pid es 9630
hay unos 20 segs para hacer en otro terminal 'pmap 9630' y ver las zonas
```

Memory management: example I

- Memory map for that process

```
$ pmap 9630
9630:   ./a.out
00005641d8cee000      4K r---- a.out
00005641d8cef000      4K r-x-- a.out
00005641d8cf0000      4K r---- a.out
00005641d8cf1000      4K r---- a.out
00005641d8cf2000      4K rw--- a.out
00005641d8cf3000 262144K rw---   [ anon ]
00005641ea585000    132K rw---   [ anon ]
00007f02b5aac000    136K r---- libc-2.31.so
00007f02b5ace000   1380K r-x-- libc-2.31.so
00007f02b5c27000    316K r---- libc-2.31.so
00007f02b5c76000     16K r---- libc-2.31.so
00007f02b5c7a000      8K rw--- libc-2.31.so
00007f02b5c7c000     24K rw---   [ anon ]
00007f02b5c9f000      4K r---- ld-2.31.so
00007f02b5ca0000    128K r-x-- ld-2.31.so
00007f02b5cc0000     32K r---- ld-2.31.so
00007f02b5cc9000      4K r---- ld-2.31.so
```

Memory management: example II

```
00007f02b5cca000      4K rw--- ld-2.31.so
00007f02b5ccb000      4K rw---   [ anon ]
00007fff2f123000    132K rw---   [ stack ]
00007fff2f15d000     16K r----   [ anon ]
00007fff2f161000      8K r-x--   [ anon ]
 total           264508K
```

Memory management: example

- Output (linux 64 bit system, statically linked)

```
$ ./a.out
variable local:      0x7ffc6d75c0f8
static inicializada:0x4b20f0
global inicializada:0x4b2100
global sin inicial.:0x4b42c0
constante:           0x487008
puntero:             0x11acb5f0
brk:                 0x11aec000
funcion:             0x401c2d
funcion libreria     0x408db0
mi pid es 9573
hay unos 20 segs para hacer en otro terminal ’pmap 9573’ y ver las zonas
```

Memory management: example

- Memory map for that process

```
$ pmap 9573
9573:   ./a.out
0000000000400000      4K r---- a.out
0000000000401000    536K r-x-- a.out
0000000000487000    156K r---- a.out
00000000004af000     12K r---- a.out
00000000004b2000     12K rw--- a.out
00000000004b5000 262148K rw---   [ anon ]
0000000010d05000    136K rw---   [ anon ]
00007ffcf1101000    132K rw---   [ stack ]
00007ffcf11d8000     16K r----   [ anon ]
00007ffcf11dc000      8K r-x--   [ anon ]
 total           263160K
```

Memory management: example

- Output (freeBSD 64 bit system)

```
$./a.out
static inicializada:0x401288
global inicializada:0x401240
global sin inicial.:0x4012c0
constante:          0x400b70
puntero:            0x800c09000
brk:                0x10402000
funcion:            0x4009c5
funcion libreria    0x4005d0
mi pid es 1302
hay unos 20 segs para hacer en otro terminal 'pmap 1302' y ver las zonas
$
```

Memory management: example I

- Memory map for that process

```
$ procstat vm 1302
  PID              START                END PRT  RES PRES REF SHD FLAG  TP PATH
 1302           0x400000           0x401000 r-x    1    4   1   0 CN--- vn /home/usuario/a.out
 1302           0x401000         0x10402000 rw-    1    1   1   0 ----- df
 1302        0x800401000        0x800407000 r--    6   29 185  57 CN--- vn /libexec/ld-elf.so.1
 1302        0x800407000        0x80041e000 r-x   23   29 185  57 CN--- vn /libexec/ld-elf.so.1
 1302        0x80041e000        0x80041f000 r--    1    0   1   0 C---- vn /libexec/ld-elf.so.1
 1302        0x80041f000        0x800442000 rw-   27   27   1   0 ----- df
 1302        0x800443000        0x8004c7000 r--   92  408 273 145 CN--- vn /lib/libc.so.7
 1302        0x8004c7000        0x800613000 r-x  292  408 273 145 CN--- vn /lib/libc.so.7
 1302        0x800613000        0x80061b000 r--    8    0   2   0 C---- vn /lib/libc.so.7
 1302        0x80061b000        0x80061c000 rw-    1    0   2   0 C---- vn /lib/libc.so.7
 1302        0x80061c000        0x800623000 rw-    7    0   1   0 C---- vn /lib/libc.so.7
 1302        0x800623000        0x80084d000 rw-   16   16   1   0 ----- df
 1302        0x800a00000        0x801200000 rw-   15   15   1   0 ----- df
 1302     0x7fffdffff000     0x7ffffffdf000 ---    0    0   0   0 ----- gd
 1302     0x7ffffffdf000     0x7ffffffff000 rw-    4    4   1   0 ---D- df
 1302     0x7ffffffff000     0x800000000000 r-x    1    1  67   0 ----- ph
$
```

Memory management: example

- Output (freeBSD 64 bit system, statically linked)

```
$ ./a.out
variable local:      0x7fffffffe988
static inicializada:0x490928
global inicializada:0x4908e0
global sin inicial.:0x494020
constante:           0x4756e0
puntero:             0x80080f000
brk:                 0x106b2000
funcion:             0x4004f4
funcion libreria     0x4009f0
mi pid es 1320
hay unos 20 segs para hacer en otro terminal 'pmap 1320' y ver las zonas
```

Memory management: example

- Memory map for that process

```
$ procstat vm 1320
  PID              START                END PRT  RES PRES REF SHD FLAG  TP PATH
 1320           0x400000           0x48f000 r-x  143  880   2   1 CN--- vn /home/usuario/a.out
 1320           0x48f000           0x493000 rw-    4    0   1   0 C---- vn /home/usuario/a.out
 1320           0x493000          0x106b2000 rw-   16   16   1   0 ----- df
 1320        0x800600000        0x800e00000 rw-   19   19   1   0 ----- df
 1320     0x7fffdffff000     0x7ffffffdf000 ---    0    0   0   0 ----- gd
 1320     0x7ffffffdf000     0x7ffffffff000 rwx    3    3   1   0 ---D- df
 1320     0x7ffffffff000     0x800000000000 r-x    1    1  67   0 ----- ph
$
```

Memory management: example

- Output (solaris 64 bit system)

```
$ ./a.out
variable local:      7fffbffff668
static inicializada:5017e8
global inicializada:5017a0
global sin inicial.:501820
constante:           400ea8
puntero:             10501830
brk:                 10505820
funcion:             401252
funcion libreria     401068
mi pid es 1608
hay unos 20 segs para hacer en otro terminal 'pmap 1608' y ver las zonas
```

Memory management: example I

- Memory map for that process

```
$ pmap 1608
1608: ./a.out
0000000000400000      8K r-x----  [ text ] /export/home/usuario/a.out
0000000000501000      4K rw-----  [ data ] /export/home/usuario/a.out
0000000000502000 262160K rw-----  [ heap ]
00007FFFBF000000   2660K r-x----  [ text ] /lib/amd64/libc.so.1
00007FFFBF399000     80K rw-----  [ data ] /lib/amd64/libc.so.1
00007FFFBF3AD000     36K rw-----  [ data ] /lib/amd64/libc.so.1
00007FFFBF400000    348K r-x----  [ text ] /lib/amd64/ld.so.1
00007FFFBF557000      4K r------  [ dtrace ] /lib/amd64/ld.so.1
00007FFFBF658000     20K rwx----  [ data ] /lib/amd64/ld.so.1
00007FFFBF65D000      4K rwx----  [ data ] /lib/amd64/ld.so.1
00007FFFBF6D0000     24K rw-----  [ anon ]
00007FFFBF6E5000     64K rw-----  [ anon ]
00007FFFBF6F6000     12K r--s---  [ anon ]
00007FFFBF6FA000      4K r--s---  [ anon ]
00007FFFBF6FC000      4K r--s---  [ anon ]
00007FFFBF6FE000      4K r-x----  [ anon ]
00007FFFBFFFD000     12K rw-----  [ stack ]
         total   265448K
```

Memory management: example

- Compile and run this C program. Then check its address space several times as the program keeps running

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <limits.h>

#define TROZO  100*1024*1024
#define PUNTO  (10*1024*1024)

void accede (char * p, unsigned long long tam)
{
  unsigned long long i;
  for (i=0; i< tam; i++){
     p[i]='a';
     if ((i%PUNTO)==0)
        write (1,".",1);  /*imprime un punto cada 10 Mbytes accedidos*/
  }
}
```

Memory management: example

```
main (int argc,char*argv[])
{
  char *p;
  unsigned long long total=0, cantidad=TROZO;
  unsigned long long maximo=ULLONG_MAX;

  if (argv[1]!=NULL){
        maximo=strtoull(argv[1],NULL,10);
        if (argv[2]!=NULL)
            cantidad=strtoull(argv[2],NULL,10);
  }
  while (total<maximo && (p=malloc(cantidad))!=NULL){
        total+=cantidad;
        printf ("asignados %llu (total:%llu) bytes en %p\n", cantidad,total,p);
        accede (p,cantidad);
getchar();

  }
  printf ("Total asignacion: %llu\n",total);
 sleep(10);
}
```

Address space F.A.Q.

- In previous slides we have seen the address space of several processes but **What is the address space of a process?**
- The address space of a process is the **set of memory addresses it can access**
- **What happens if a process acesses a memory address outside of its address space?** IT CAN NOT (That's the very definition of address space)
- **What happens if a process TRIES TO ACCESS a memory address outside of its address space?** The hardware won't allow it, notify the Operating System and then, the Operating System would probably have the process killed with a *segmentation fault* type of error
- **What is the user address space of a process?** the set of memory addresses it can access when in *user mode*

Address space F.A.Q.

- **Why is not the address space of a process the same as the memory on the machine it's in?**
- several reasons
  - there are more processes on that machine
  - on modern systems the address of the processes are **fake** (although the term *logical* or *virtual* is more adecuate)
  - the address space of a process probably needs higher addresses than those that there exist on the machine it is in
  - the address space of a process might be bigger than the memory the machine has installed
  - the program would be specific to that machine
  - ...

Address space F.A.Q.

- **How is the address space of a process structured?** In zones (usually called segments or regions), each one of them has a different part of the process: code, data, stack . . .
- **Can the address space of a process grow?**
- yes, of course
  - the process allocates memory (for example with *malloc*): the data segment grows (in linux if you allocate more tha 128K, a new region is created)
  - the process maps a file: a new region for the mapped file is created
  - the process calls a recursive function, or a function with many (or large) local variables: the stack grows
  - the process allocates shared memory: a new region for the shared memory is created

Address space F.A.Q.

- **Can the address space of a process shrink?**
- yes, of course. Deallocating items usually makes the address space smaller
- **Is there any command that shows the address space of a prcess?**
- Yes: *pmap* in linux and solaris, *procsat vm* in FreeBSD, *procmap* in OpenBSD, *vmmap* en MacOS . . .
- **Can the address space of a process hav 'holes' in it or does it need to be contiguous?**
- It can have holes, in fact some of the regions in it are necessarily not contiguous as there must be room left between them to allow for region growth

Swap

- The *Swap* area is a part of the secondary storage (disk) used as auxiliar memory
- A running process needs to be in memory. Using swap can increase the multiprogramming level,
    - If a process in the swap zone is selected by the scheduler, the process needs to be loaded in memory, which increases the context switch time.
    - To swap processes that are waiting for I/O to be completed (pending I/O), the OS must transfer I/O to the system buffers in kernel space and then to the I/O device. This also adds overhead.
    - For these reasons, modern Operating Systems usually swap pages and rarely swap whole processes.

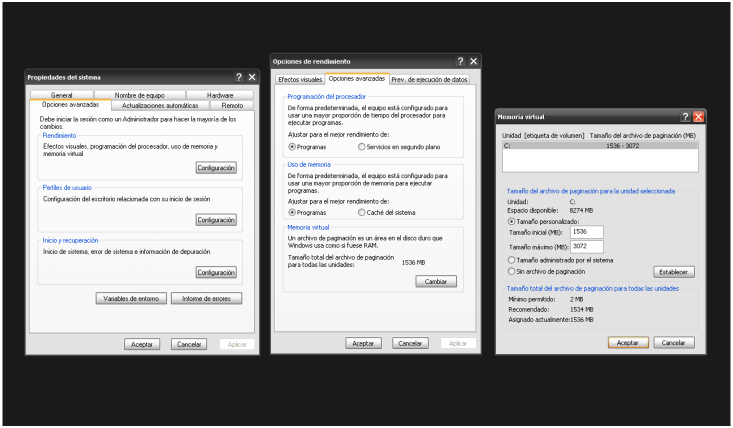
Swapping vs paging

- Writing **whole processes** to the swap device was use in old (obselete) systems to increase the multiprogramming level.
    - Those were called *swapping systems*
- Modern systems have virtual memory: they write to the swap device pages of the processess rather than **complete processes** .
    - We call them *paging systems*
- In virtual memory systems we get illusion of nearly unlimited memory

Swap

- The swap area can be a dedicated disk, partition on a disk or a file in the file system.
- Using a file as swap device is a more flexible solution, its location and size can be changed easily.
- Using a file as swap device is less efficient because it uses the indirections of the file system to access the data.
- MS Windows systems use a swap file.
- Unix type systems use swap partitions, although swap files can be configured for these systems.

Swap file in MS Windows



Propiedades del sistema

General | Nombre de equipo | Hardware | Opciones avanzadas | Actualizaciones automáticas | Remoto

Debe iniciar la sesión como un Administrador para hacer la mayoría de los cambios.

Rendimiento
Efectos visuales, programación del procesador, uso de memoria y memoria virtual

Perfiles de usuario
Configuración del escritorio relacionada con su inicio de sesión

Inicio y recuperación
Inicio de sistema, error de sistema e información de depuración

Opciones de rendimiento

Efectos visuales | Opciones avanzadas | Prev. de ejecución de datos

Programación del procesador
De forma predeterminada, el equipo está configurado para usar una mayor proporción de tiempo del procesador para ejecutar programas.
Ajustar para el mejor rendimiento de:
[✓ Programas] Servicios en segundo plano

Uso de memoria
De forma predeterminada, el equipo está configurado para usar una mayor proporción de memoria para ejecutar programas.
Ajustar para el mejor rendimiento de:
[✓ Programas] Caché del sistema

Memoria virtual
Un archivo de paginación es un área en el disco duro que Windows usa como si fuese RAM.
Tamaño total del archivo de paginación para todas las unidades: 1536 MB

Memoria virtual

Unidad [etiqueta de volumen]	Tamaño del archivo de paginación (MB)
C:	1536 - 3072

Tamaño del archivo de paginación para la unidad seleccionada
Unidad: C:
Espacio disponible: 8274 MB
[✓ Tamaño personalizado:]
Tamaño inicial (MB): 1536
Tamaño máximo (MB): 3072
Tamaño administrado por el sistema
Sin archivo de paginación

Tamaño total del archivo de paginación para todas las unidades
Mínimo permitido: 2 MB
Recomendado: 1534 MB
Asignado actualmente: 1536 MB

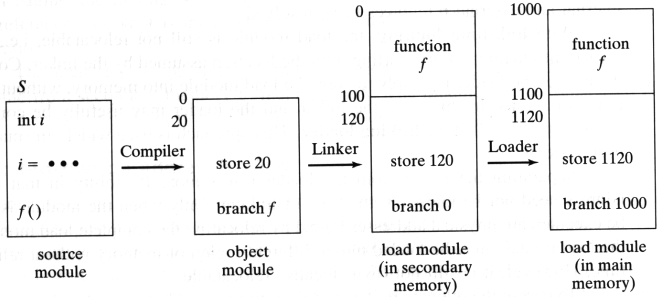
Memory management: relocation

- We start with source code $- >$ (compilation) $- >$ object code
- Several object code files $- >$ (linking) $- >$ executable file
- Executabe file $- >$ (load and execution) $- >$ process in memory
- Source code $- >$ executable file $- >$ process in memory
- In the source code there are variables, functions, procedures, ...
- In the process in memory there are contents of memory addresses, jumps to addresses that contain code...
- When and where are these transformations done?

Memory management: relocation

- **Absolute code**: Addresses are obtained at compilation (and/or linking) time (example: MS-DOS .COM files)
  - At compilation/linking time it is necessary to know the addresses for execution of the program
  - Lack of portability of the executable file. It can not run in other memory locations.
- **Static relocation**: Addresses are obtained when the program is loaded in memory (the executable file contains relative references) (example: MS-DOS EXE files)
  - After loaded in memory, the program can not be moved to other memory location
  - Swapping is possible only if processes return to the same memory positions they used before being swapped out (fixed partitions)

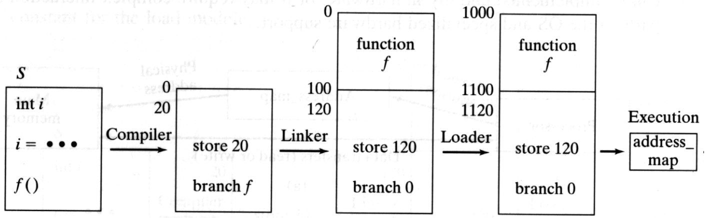
Static relocation



Memory management: relocation

- **Dynamic relocation**: Addresses are obtained at execution time. The running process uses memory addresses that are not real physical memory positions. Translation is done in execution time by specialized hardware (example: MS XP Windows EXE files)
  - No restrictions to swapping. Swapped processes can be swapped in memory in any memory location.
  - Distinction between *Virtual or Logical address space* and *Physical address space*.
  - It is necessary hardware that translates logical addresses in physical addresses.
- Modern systems use dynamic relocation
- With dynamic relocation, linking can be postponed to execution time. Dynamic linking (MS Windows DLLs, lib*.so in linux).

Dynamic Relocation



Protection

- Memory must be protected
  - A process can not directly access the OS memory
  - A process can not access memory of other processes
- Simplest hardware to support protection
  - Two limit registers
  - One base (relocation) register and one limit register

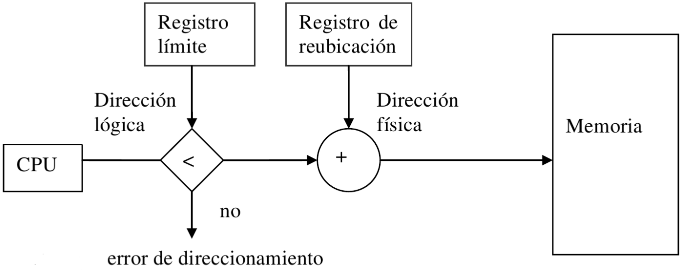
Protection

- Two limit registers
  - Every address generated by a running process must be in the range of the values stored in the limit registers. Otherwise an exception is produced.
  - The hardware must provide these limit registers.
  - The values of the registers are updated in a context swicht and stored in the Process Control Block.
  - Changing the values of these registers is a privileged instruction (kernel mode)

Protection and relocation

- With base (relocation) and limit register
  - Base register contains value of lowest physical address the process can access. Limit register contains range of logical addresses. Each logical address must be less than the limit register (otherwise, an exception is produced), which is added to the address contained in the base register.
  - The hardware must provide these base and limit registers.
  - The values of the registers are updated in a context swicht and stored in the Process Control Block.
  - Changing the values of these registers is a privileged instruction (kernel mode)
  - This hardware supports protection and is in fact s way of doing dynamic relocation.

Base and limit registers



Protection and relocation

- With base (relocation) and limit register
    - The program has the illusion of running on a dedicated machine, with memory starting at address zero and having *lim* bytes available
    - This is in fact a segmented system with only one segment.

Protection

- In modern systems memory protection is supported by the addressing mechanisms.
- Segmentation and paging provide effective memory protection and relocation.
- It is necessary at least two execution modes: user mode and kernel or system mode.

Simple schemes (obsolete)

No multiprogramming systems
Multiprogramming Systems

Simple schemes: No multiprogramming systems

- This approach has been obsolete for a long time
- In Operating Systems without multiprogramming there were only two memory areas: one for the OS and one for the user process.
- Typically the OS in the low positions of memory and the rest for the user process. Concept of simple monitor (example IBSYS for IBM 7094, late 1950's)
- First generation of personal computers: OS the upper part of memory (in ROM) and the rest for user processes (example: zx spectrum 1982)
- OS in the lower memory addresses but with some parts of it in the upper part. Example: First versions of MS-DOS, IBM PC, 1981

Simple schemes (obsolete)

No multiprogramming systems
Multiprogramming Systems

Memory management: Simple schemes

- For multiprogrammed OS the simplest scheme is to split the memory in partitions with a process in each partition.
- Two alternatives
  - Fixed size partitions: Allows for a fixed number of processes in memory
  - Variable size partitions: The number and size of partitions can vary

Memory management: Simple schemes

- Fixed size partitions
  - Internal and external fragmentation
  - Used in IBM OS/360 MFT (Multiprogramming with a Fixed number of Tasks)
- Variable size partitions
  - Negligible *internal fragmentation*, but also suffers from *external fragmentation*
  - Compactation of memory to solve external fragmentation. Very high cost.
  - Used in IBM OS/360 MVT (Multiprogramming with a Variable number of Tasks)
- Again, this schemes have been obsolete for a long time. IBM delivered its OS/360 in 1967

Memory management: Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
    - A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays, etc
- This has also become obsolete. Last machine to use segmentation was IBM OS/2 1.3 on intel's 286 hardware (around 1990)

Memory management: Segmentation

- The address space of a process has variable size blocks called *segments*
- A logical address is composed of a segment (or *segment number*) and an *offset* inside the segment, <segment-number, offset>
- The segment number is the entry number in the Segment Table for that process. Each entry of the Segment Table contains the Base Address, i.e. the starting physical address for the associated segment, and the segment size (Limit).
  - Segment-table base register (STBR) (in the processor) points to the segment table's location in memory

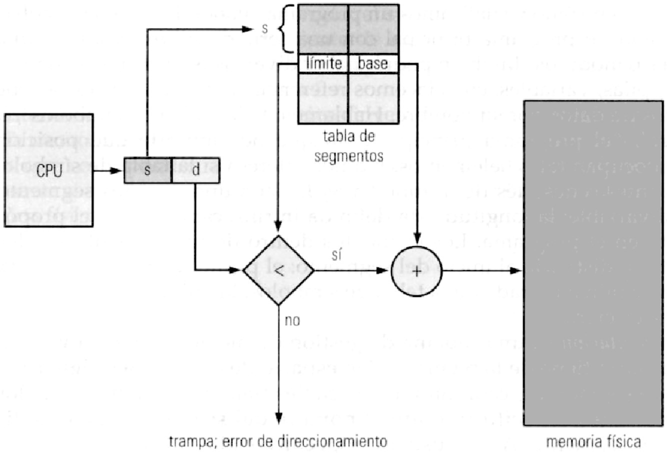
Memory management: Segmentation

- Changing the values of these registers in a context switch is a privileged instruction. The values are stored in the Process Control Block.
- logical address: $<$segment-number, offset$>$
- physical address: Base Address + offset
- if the offset is less than the limit, the physical address is obtained adding the offset to the Base Address, otherwise an addressing error (exception) is produced and control goes to the OS

Memory management: Segmentation

- Protection, with each entry in the segment table associate:
  - validation bit (legal/illegal segment)
  - read/write/execute privileges
  - kernel/user mode accesible segment
  - ...
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Memory management: Segmentation



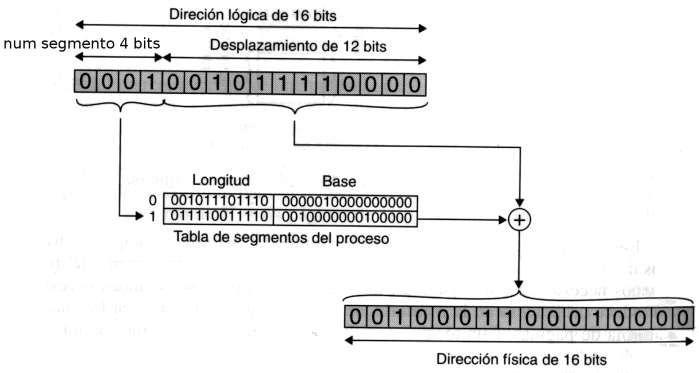
Segmentation: example

- Let us consider a system with segmentation with the following properties:
  - logical addresses of 16 bits (4 bits for the segment number, 12 bits for the offset)
  - each entry of the segment table has 28 bits, the 12 most significant for the limit and the 16 least bits for the base address
  - A process has 2 segments and the first two entries in the segment table of the process contain the values 0x2EE0400 and 0x79E2020 respectively
    - What physical address corresponds to a reference to logical address 0x12F0?
    - What physical address corresponds to a reference to logical address 0x0342?
    - What physical address corresponds to a reference to logical address 0x021F?
    - What physical address corresponds to a reference to logical address 0x190A?

Memory management: Segmentation example

- logical address 0x12F0, physical address 0x2310
- a reference to the logical address 0x0342 causes an addressing error
- logical address 0x021F, physical address 0x061F
- a reference to the logical address 0x190A causes an addressing error
- Remeber, this is an OVERSIMPLIFIED example: 4:12 bits is unrealistic. Also there are more things in the segment table than the base address and limit (r/w, k/u . . .)

Memory management: Segmentation



Fragmentation in segmentation systems

- *Internal fragmentation*.
  - The segment size is a multiple of a fixed number of bytes (for example 16 bytes), therefore allocated memory may be slightly larger than requested memory; this size difference is memory internal to the segment, but not being used. Negligible
- *External fragmentation*.
  - Segments are variable size memory blocks. After assigning and releasing memory, holes (blocks of available memory) of various sizes are scattered through memory.
  - External fragmentation: total memory space exists to satisfy a specific request, but it is not contiguous. Compactation solves the problem at the expense of computional cost.

Segmentation: dynamic storage-allocation problem

- The OS accounts for both the assigned and free memory. After releasing a block of memory, adjacent free blocks are collapsed into a larger free block.
- How can the OS satisfy a request of size *n* from a list of free holes?
    - **first fit** Allocate the first hole that is big enough. Fast. Small holes appear in low areas of memory and large holes in high areas, assuming the search for holes start in the low areas.
    - **next fit** Allocate the next hole large enough, searching from the last allocated block
    - **best fit** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
    - **worst fit** Allocate the largest hole; must also search entire list. Produces the largest leftover hole
- First-fit and best-fit are found to operate better than worst-fit in terms of speed and storage utilization

Segmentation: amount of memory in holes

- In a system with segmentation, given s the average size of a segment and $k = \frac{average\_size\_of\_a\_hole}{average\_size\_of\_a\_segment}$

$$\frac{memory\_in\_holes}{total\_memory} = \frac{k}{k+2}$$

- for n segments, the amount of memory in segments is ns.
- two adjacents holes collapse into a single hole, therefore there are double number of segments than holes. For $\frac{n}{2}$ holes the amount of memory in holes is $\frac{n}{2}ks$
- therefore the rate is

$$\frac{memory\_in\_holes}{total\_memory} = \frac{\frac{n}{2}ks}{\frac{n}{2}ks+ns} = \frac{k}{k+2}$$

Memory management: Segmentation

- *Hardware* support is needed
- It is a form of dynamic relocation
- It enables memory protection
- Sharing data or code segments is possible
- Logical (virtual) address speace and physical address space need not be the same size.

Segmentation implementation: Intel 8086

- Intel 8086, 4 segments with a segment register for each segment (code CS, data DS, stack SS, extra ES)
- Rudimentary: no segment tables in memory.
- Rudimentary: no memory protection; any program could access any memory area.
- 32 bit virtual addresses (16bits segment, 16 bits offset).
- 1 Megabyte physical (20 bits physical address space)
- 32 logical addresses segment:offset yielded a 20 bit physical address $segment << 4 + offset$

Segmentation Implementations:Intel 286

- Intel 80286: 1 Gigabyte addressable virtual memory in 64k segments
- 32 bit logical addresses (16 bits segment, 16 bit offset).
- 24 bits physical addresses (16 megabytes physically addressable)
- Segment tables in memory, pointed by a processor register.
- Although address were 32 bit, virtual address space was 1Gb: of the 16 bit segment, 2 bits were for privilege level, 1 bit for segment table selection and 13 bits for segment number. That yielded 2^{13} segments in each of the two possible segment tables, that's to say, a total of 2^{14} segments. Total virtual address space: $2^{14} * 2^{16} = 2^{30}$ (1Gb)
- MS Windows 3.1 in standard mode used this segmentation mechanism. Windows 3.1 had an *"Enhanced"* mode to operate with i386 processor paging.
- Intel 386 used Paged Segmentation

Memory management: paging

- Physical address space of a process can be noncontiguous; process can be allocated physical memory wherever available
- Avoids external fragmentation, still has internal fragmentation
- Avoids problem of varying sized memory chunks
- Enables dynamic relocation, protection and sharing of memory

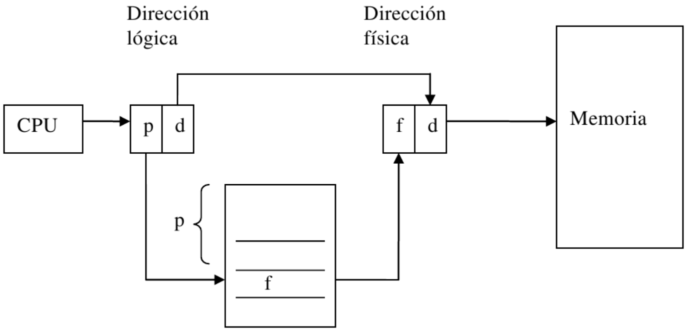
Memory management: paging

- Divide physical memory into fixed-sized blocks called *frames* (sometimes physical pages). Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called pages
- Logical (virtual) address space and physical address space need not be the same size.
- The OS keeps track of all free frames. To run a program of size N pages, the OS needs to find N free frames and load program in them
- The OS sets up a page table to translate logical to physical addresses. A processor register (Page Table Base Register) contains the base address of the page table for a process. Changing the value of this register in a context switch is a privileged instruction (kernel mode). The value is stored in the Process Control Block, and loaded again when the process is scheduled to run

Memory management: paging

- Address generated by CPU is divided into:
- Page number (p) used as an index into a page table which contains base address of each page (frame number) in physical memory
- Page offset (d) combined with base address to define the physical memory address that is actually accessed

Memory management: paging



Paging: Example

- Consider a system with 16 bits logical addresses, 7 most significant bits for the page number and the 9 least significant bits for the offset in the page.
- Page size is 512 bytes $= 2^9$
- A process makes a reference to a memory address 0x095f (0000 1001 0101 1111)
- This is a reference to page number 4 and page offset 0x15f

Paging: Example

- In the entry for page 4 in the page table, we can get the physical address for this page
- Let us assume that the Base Address of the Physical Page is 0xAE00 (1010 1110 0000 0000)
- Therefore the final physical address is es 0xAF5F (1010 11111 0101 1111)
- As all frames start at offset 0, in the page table we only get the most significant bits of the frame address (without the offset part), that would be '1010111' in this example, which is also the frame number

Paging: A more detailed example

- Logical and real space addresses need not be the same size
- Let us consider a system with 16 bits logical addresses and 20 bits physical addresses, with a page size of 512 bytes (2^9)
- The first 7 bits of a logical address are the page number, and the remaining 9 bits is the page offset (page size is 512 bytes)
- A page table must have (2^7) entries (one for each page).

Paging: A more detailed example

- 4 bytes (32 bit) each page table entry is a reasonable value as the frame address needs 11 bits, and the remaining 21 bits are enough for presence bit, reference bit, r/w, privilege level . . . . Even more, an entry of 4 bytes would make the page table fit in exactly one page, which is a very convenient design issue.
- Let us also assume that the format of a page table entry is as follows:
    - the 11 most significant bits of a page table entry are the physical frame number
    - bit 0 is the presence bit
    - bit 1 the reference bit
    - bit 2 r/w (0 for read only)
    - bit 3 privilege level (1 kernel mode)
    - the rest of the bits are reserved for O.S. use.

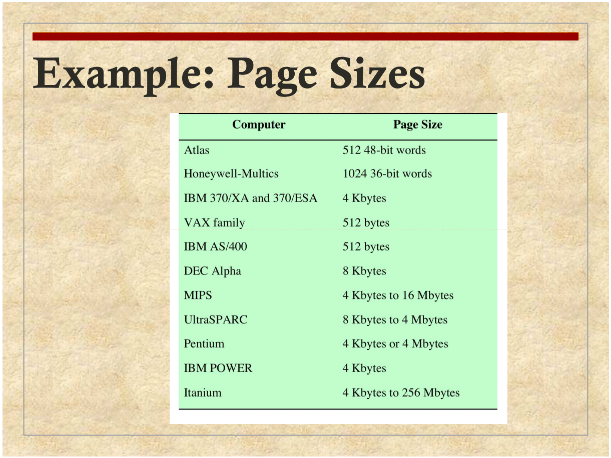
Paging: A more detailed example

- A process tries to read from memory address 0x194d (0001 1001 0100 1101)
- This is a reference to page number 12 (0xC) and page offset 0x14d
- Should the value of entry 0xc in the process page table be 0x24800001 (0010 0100 1000 0000 0000 0000 0000 0001). The aforementioned read reference would get the contents of physical memory address 0x2494d (0010 0100 1001 0100 1101)
- If the process tries to write to memory address 0x183f, it would produce an exception (and the O.S. would probably have it killed), as this is a reference to offset 0x3f in page 0xc, and page 0xc is marked readonly (bit 2 of the page table entry-0x24800001-is 0, page is readonly)

Paging

- With paging the memory for a process need not be contigous
- The page size is defined by the hardware, for example:
  - The Intel x86 32 bits architecture has a page size of 4 Kbytes
  - The Sparc architecture has a page size of 8 Kbytes
- No external fragmentation
- Internal fragmentation
- The larger the page sizes the larger the internal fragmentation will be
- Smaller page sizes mean more entries in the page tables for processes, so more memory is lost in page tables

Page sizes for different architectures



Example: Page Sizes

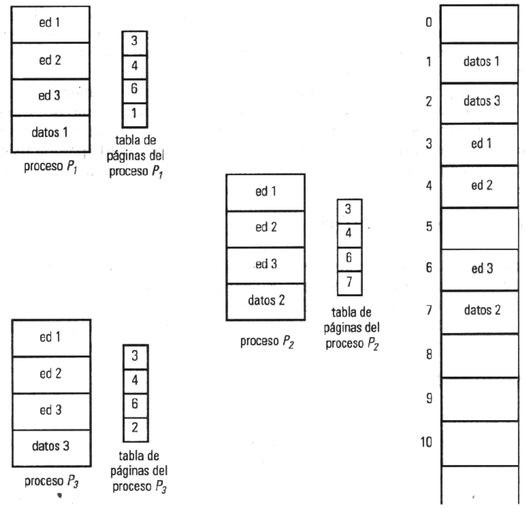
Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Paging

- The OS must account for: the free physical pages (frames) and the frames assigned to processes
- The OS manages the table pages of different process in the context switch
- Processes' view of the memory and physical memory are now very different
- Memory Protection: because of the way paging is implemented, a process can only access its own memory following its page table, and its Page Table can only be changed by the OS
- Paging allows memory sharing among processes: the associated entries in their page tables point to the same physical pages

Memory management: paging



Paging

- Hardware support is needed (for example no paging on intel 286, paging available on i386)
- Each entry of the PT has other information beside the address of the physical page: presence bit (the page is in memory) , access bit (1 page was accessed, but can be 0 for a accessed page after a bit reset), dirty (modified page) bit, read only, read write, privilege level (kernel only, user), etc.

Paging: Implementation of the Page Table

- In theory, two different ways to implement the page tables could be used
    - Dedicated processor registers
    - In memory
- All present machines use page tables in memory

Paging: Implementation of the Page Table

- **Dedicated registers:** The processor has registers to store the page table of the running processes.
  - In a context switch the page table is stored in the Process Control Block, and the registers are loaded with the page table of the new running process.
  - Address translation is fast because the page table is in the processor registers
  - High cost because many processor registers are needed, for this reason it is not used in modern systems
  - Slow context switch because many registers are implied

Paging: Implementation of the Page Table

- **In memory:** Page tables are kept in main memory.
  - The Page-table base register (PTBR) points to the page table of the running process. In a context switch the value of this register is stored in the Process Control Block, and the register is loaded with the page table base address of the new running process.
  - Address translation is slow because every data/instruction access requires two memory accesses: one for the page table and another for the data/instruction
  - Fast context switch because only one register is implied
  - The two memory access problem can be solved by the use of a special fast-lookup hardware cache (associative memory) called translate look-aside buffers (TLBs)

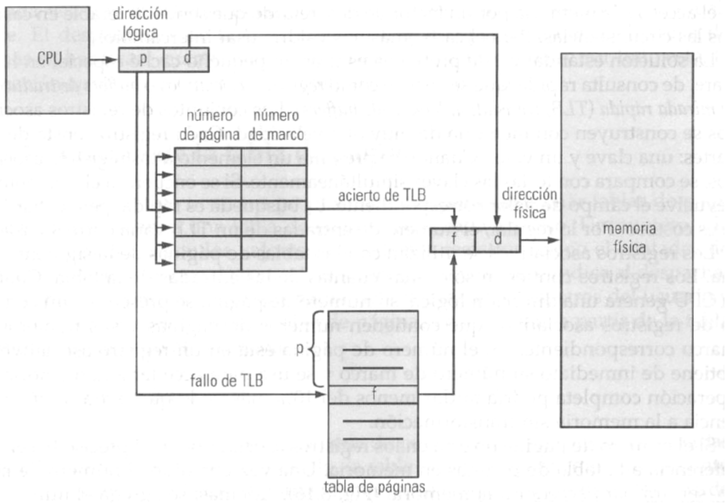
Paging: Implementation of the Page Table

- The TLB is an associative memory that performs a parallel search, i.e., the TLB contains pairs <number of logical page, frame number>.
- Given a logical address (p, d), If p is in associative register contained in the TLB, get frame number out, otherwise get frame number from page table in memory

Paging: Implementation of the Page Table

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry. This uniquely identifies each process to provide address-space protection for that process. Otherwise theres a need to flush it at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
    - Replacement policies must be considered
    - Some entries can be wired down for permanent fast access

Memory management: Paging with TLB



Paging: Effective access time with TLBs

- Let's consider that TLB Associative Lookup time is ϵ (time unit). Can be $<$ 10% of memory access time
- We define the hit ratio (α) as the percentage of times that a page number is found in the associative registers. The bigger the number of associative registers the highest the hit ratio will be
- The Effective Access Time for a memory with access time T is
  - $EAT = \alpha(T + \epsilon) + (1 - \alpha)(2T + \epsilon)$
- Consider the following example $\alpha = 80\%$, $\epsilon = 20ns$ for TLB search, 100ns for memory access
  - $EAT = 0.80x120 + 0.20x220 = 140ns$ (instead of 200 ns without TLB)
- The more realistic example where $\alpha = 99\%$, $\epsilon = 20ns$ for TLB search, 100ns for memory access, yields
  - $EAT = 0.99x120 + 0.01x220 = 121ns$ (instead of 200 ns without TLB)

Inverted page tables

- in systems with virtual memory, there are many pages of the processes that are not loaded in physical memory; however the page tables of each process have to be large enough to accommodate the whole address space of the process
    - A 32 bits address space using 4K pages would need 2^{20} pages. The table page for a process would have to have 2^{20} entries. Assuming 4 bytes entries, each page table would need 4 Mb
- Two solutions
    - **multilevel paging** (mixed systems)
    - **inverted page tables**
        - one entry for each physical frame
        - each entry has information about the page contained in that frame (proccess and logical address)
        - To optimice the search time usually a *hash* function is used
        - Used in the IBM PowerPC and IBM AS/400 architectures

Inverted page table

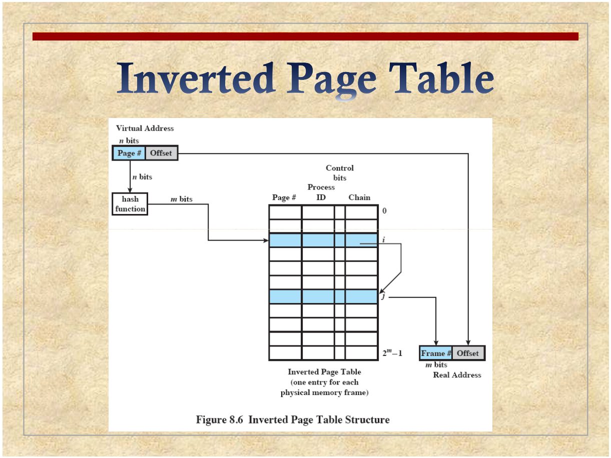


Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Two level page table

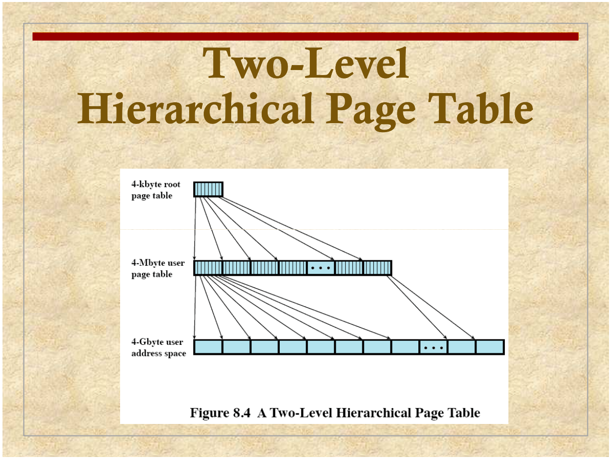

Two-Level
Hierarchical Page Table

Figure 8.4 A Two-Level Hierarchical Page Table

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Mixed systems

- A combination of paging and/or segmentation
- As paging provides better memory usage, there's always paging as the last managing system
  - **segmented paging**: obsolete systems, used in IBM system 370 (around 1972), the page table of a proccess was segmented
  - **paged segmentation** the segments are paged
  - **multilevel paging**: there are various levels of paging
- Today's systems use more sophisticated schemes. We will see:
  - intel x86 32bits PC archictecture: paged segmentation with two level paging
  - intel x86 64bits PC architecture: four level paging

Mixed systems

Advantages of multilevel paging

Advantages of multilevel paging

- Multilevel paging is, a priori, more complex and slower than single level paging (although ultimately it will depend also on memory access times and the workings of both cache and TLBs)
- What might be the advantages of multilevel paging vs single level?
- Let's asume a machine with 16bit_physical/16bit_logical address space and 4K pages.
- Remember, this example is **OVERSIMPLIFIED**:
  - physical and logical address spaces need not be the same size
  - a 4K page is way too large for a 16 bit adress space
  - today a 16 bit machine is completely obsolete

Advantages of multilevel paging

- Although this example is **NOT REAL**. It can show us the advantages of multilevel paging
- We'll see how processes page tables look in two cases
  - single level page table: four bits page number, 12 bits offset
  - two level page table: two bits first level page number, two bits second level page number, 12 bits offset **OVERSIMPLIFIED and exaggerated: two bits for each evel page number???**

Advantages of multilevel paging. One level page table

- In this first two examples we use a one level page table, with 16 entries (remember 4 bits for page number makes for 16 posible pages). Also remember this is not real, page size is too big for address space, just an oversimplified example.
- For a process with big address space (a 64 K process) all entries in the page table are used
- For a process with small address space (a 16K process) only four entries are used, although the other entries are present in the page table to show that those pages do not exist in the process address space.
- The following two figures show both processes with their corresponding page tables

Example One Leve PT. Big Address Space

F	Frame 1
E	Frame 0
D	Frame 5
C	Frame F
B	Frame 9
A	Frame 8
9	Frame 7
8	Frame 6
7	Frame E
6	Frame C
5	Frame B
4	Frame A
3	Frame 2
2	Frame D
1	Frame 4
0	Frame 3

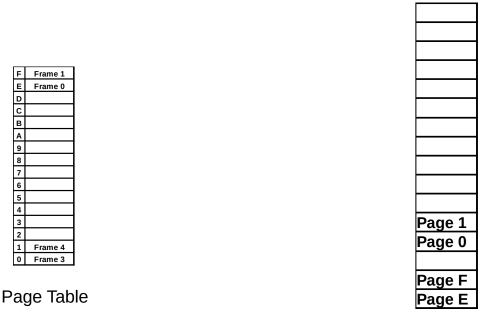
Page Table

Page C
Page 7
Page 2
Page 6
Page 5
Page 4
Page B
Page A
Page 8
Page 9
Page D
Page 1
Page 0
Page 3
Page F
Page E

Physical Memory

ONE LEVEL PAGE TABLE FOR PROCESS WITH A BIG ADDRESS SPACE

Example One Leve PT. Small Address Space



Page Table

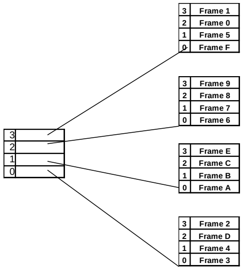
Physical Memory

ONE LEVEL PAGE TABLE FOR PROCESS WITH A SMALL ADDRESS SPACE

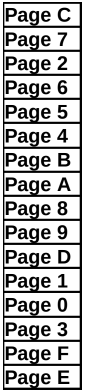
Advantages of multilevel paging. Two level page table

- Let's now consider a two level paging for the same (hypothetical) architecture: 2 bits firts level page number, two bits second level page number and 12 bits page offset
- Again, let's enphasize that this is an OVERSIMPLIFIED example (it would make no sense 4k pages in a 16 bits address space, with two bits page numbres!!)
- For the process with a big addres space
  - Access would be, in principle, slower (three memory accesses, instead of two), but that can be overcomed with the adecuate TLBs
  - The page table is bigger, 20 entries instead of 16, as can be seen in the figures
- For the small address space process, the page table is smaller, because **for pages not present in the address space, the second level table is not allocated**

Example One Leve PT. Big Address Space



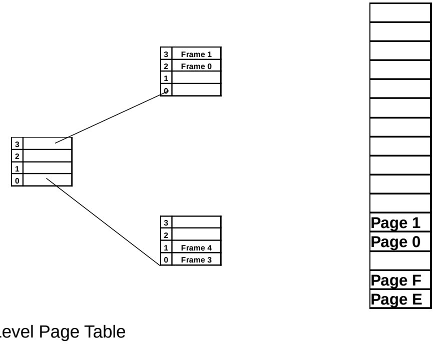
Two Level Page Table



Physical Memory

TWO LEVEL PAGE TABLE FOR PROCESS WITH A BIG ADDRESS SPACE

Example One Leve PT. Small Address Space



TWO LEVEL PAGE TABLE FOR PROCESS WITH A SMALL ADDRESS SPACE

Advantages of multilevel paging. Real world

- The previous examples were made for an OVERSIMPLIFIED hypothetical architecture, but we can see the idea behind it.
- In the proposed exercises we have one that calculates (for the Intel 32 bit architecture described in the next chapter), the size of the page table of a process for which the command pmap reports the following address space.

```
 antonio@abyecto:~$ pmap -x 3709
3709: ./shell
Address Kbytes RSS Dirty Mode  Mapping
08048000     4   4     0 r---- shell
08049000   636 572     0 r-x-- shell
080e8000   236 128     0 r---- shell
08124000     8   8     4 r---- shell
08126000     8   8     8 rw--- shell
08128000    76  12    12 rw--- [ anon ]
091c9000   136   8     8 rw--- [ anon ]
b7f11000    16   0     0 r---- [ anon ]
b7f15000     8   4     0 r-x-- [ anon ]
bfd4f000   132  16    16 rw--- [ stack ]
-------- ------- ------- -------
total kB  1260 760    48
```

Advantages of multilevel paging. Real world

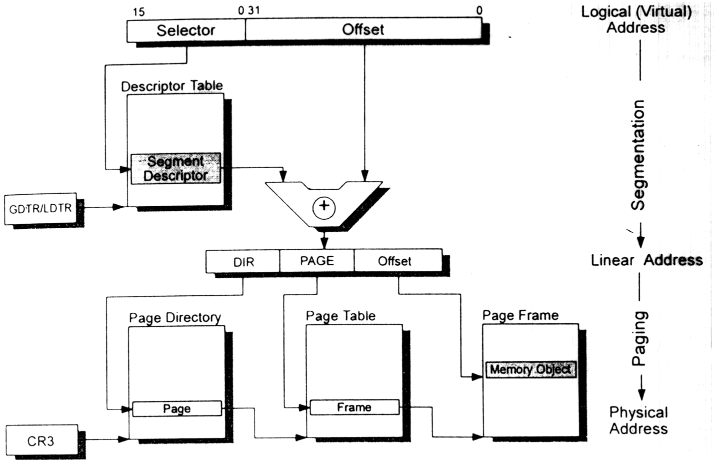
- The exercise calculates the page table size to be 20Kb.
- If the page table were to be one level page table
  - it would need 2^{20} entries (there are 20 bits for page number)
  - each entry would need at least 32 bit (4 byte) (20 bits are needed for the physical frame, plus presence, read/write, kernel/user . . .)
  - This would make the page table 4Mb in size!!

Real examples: Intel's 32 bit and 64 bit architectures

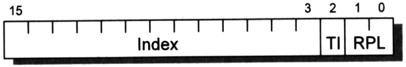
Memory management in the 32 bit PC architecture

- As an example of *actual* memory management scheme we'll look at the 32 bit PC architecture: segmentation with two levels of paging
- Each addres is comprised of
  - **selector** (16 bits) 13 bits for the segment number, 1 bit to select table(GDT-Global Decriptor Table or LDT-Local Descriptor Table) and 2 bits for the privilege level
  - **offset** 32 bits
- The 13 bits of the segment number (*selector*) serve as an index in a segment table (*descriptor table*) where we get, among other things,
  - a 32 bits base address
  - 20 bits limit (which represents a 32 bit addressing if page granularity is selected)

Memory management in the 32 bit PC architecture

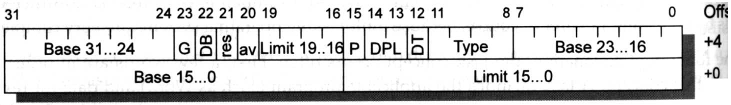


32 bit PC: Segment selector



Index:	Index into GDT or LDT
TI:	Table Indicator
	0=GDT 1=LDT
RPL:	Requested Privilege Level
	00=0 01=1 10=2 11=3

32 bit PC: Segment selector



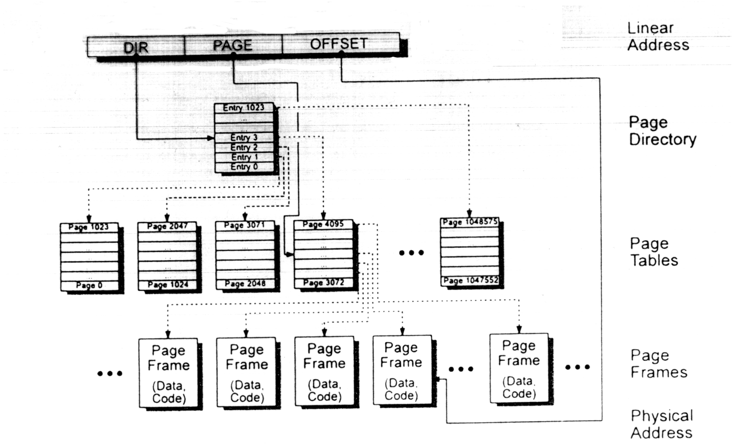
Base 31...24, Base 23...16, Base 15...0: Segment Base
Limit 19...16, Limit 15...0: Segment Limit

G:	Granularity
	0=Byte Granularity 1=Page Granularity
DB:	Segment Size (Default/Big)
	0=16-bit Operands/Addresses (Default) 1=32-bit Operands/Addresses (Big)
r:	Reserved
av:	Available for Operating System
P:	Segment Present
	0=Segment Not in Memory 1=Segment in Memory
DPL:	Descriptor Privilege Level
	00=0 01=1 10=2 11=3
DT:	Descriptor Type
	0=System Segment 1=Application Segment
Typ:	Type of System or Application Segment

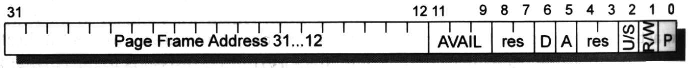
Memory management in the 32 bit PC architecture

- the 32 bit base address is added to the 32 bit offset yielding a 32 bit *linear address*
  - The first 10 bits correspond to an entry in the *page directory* table, where we get, among othe things, the address of a page table
  - The next 10 bits represent an index in the page table we got in the previous step, where we get the physical address of a page frame
  - The next 12 bits represent an offset in the aforementioned page frame
- The pages are sized 4K
- Each page table has 1024 four bytes entries (its format can be seen in one of the following figures). Each page table occupies one page

32 bit PC: linear address



32 bit PC: page table entry



Page Frame Address: address bits 31...12 of the page frame

AVAIL:	available for the operating system	
D:	dirty	
	0=page has not yet been overwritten	1=page has been overwritten
A:	accessed	
	0=page has not yet been accessed	1=page has already been accessed
U/S:	page access level (user/supervisor)	
	0=supervisor (CPL=0..2)	1=user (CPL=3)
R/W:	write protection (read/write)	
	0=page is read-only	1=page can be written to
P:	page present	
	0=page is swapped	1=page resides in memory
res:	reserved (equal 0)	

Real examples: Intel's 32 bit and 64 bit architectures

Intel's 32 bit PC architecture
Intel/amd 64 bit PC architecture

Memory management in the 64 bit PC architecture

- We will take the intel's core i7 as the example of this architecture. As of now it is not really 64 bit. Actually for the core i7 it is 48 bit logical and 52 bit physical.
- What happens with the remaining bits?
- The 16 most significant bits of a virtual address (remember, only 48 bits are meaningful) are either all 0's o all 1's. They are equal to the most significant bit of the 48 bit virtual address

Memory management in the 64 bit PC architecture

- This means that the 48 virtual address space is divided into two halves of 64bit addresses
  - lower half addresses: `0x000000000000` to `0x7fffffffffff`, which go from (in 64 bit form) `0x0000000000000000` to `0x00007fffffffffff`
  - upper half addresses: `0x800000000000` to `0xffffffffffff`, which go from (in 64 bit form) `0xffff800000000000` to `0xffffffffffffffff`
  - addresses between `0x00007fffffffffff` and `0xffff800000000000` are deemed not valid

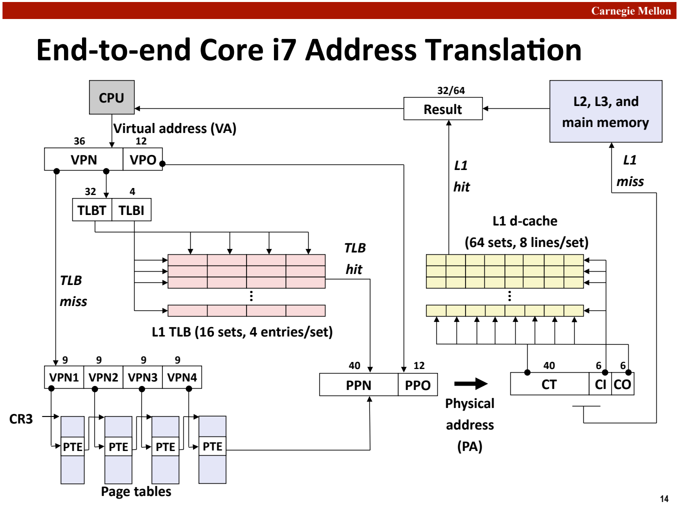
Memory management in the 64 bit PC architecture

- The 48 bit virtual address space is organized as a four level page,
  - 12 bits offset (Virtual Page Offset), making page size 4k
  - 36 bits Virtual Page Number with four level of pages, each of which has 9 bits
- This 9 bits allow for 2^9 (512) entries in each table. As entries are 64 bits (8 bytes), each table is 512x8=4096 bytes (which is exactly one page!!)

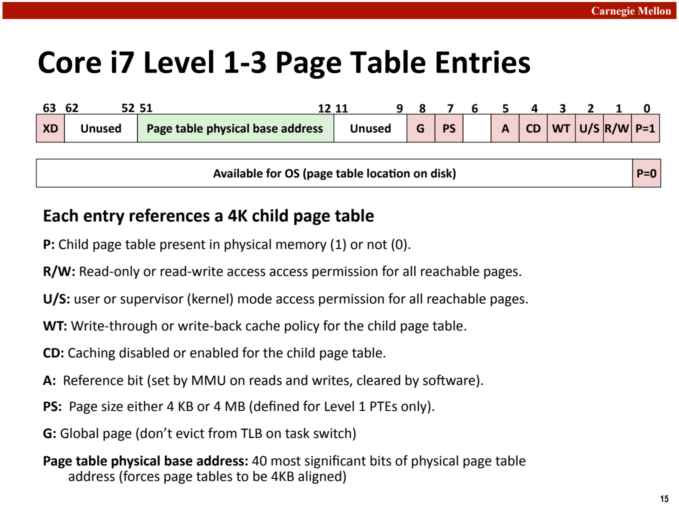
Memory management in the 64 bit PC architecture

- To speed up things, the 36 bits Virtual Page Number is fed to the TLB (32bit tag, 4bit index) that would yield (should there be a TLB hit) the 40 bit Physical Page Number.
- This PPN together with the Physical Page Offset (which is the same as the Virtual Page Offset) constitutes the 52 bits Physical Address
- The 52 Physical Address is searched in a L1 cache (40 bit tag, 6 bit index, 6 bit offset)
- The schematics can be seen in the following figures

Core i7 Address Translation



Core i7 Page Table Entries



Carnegie Mellon

Core i7 Level 1-3 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

Each entry references a 4K child page table

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

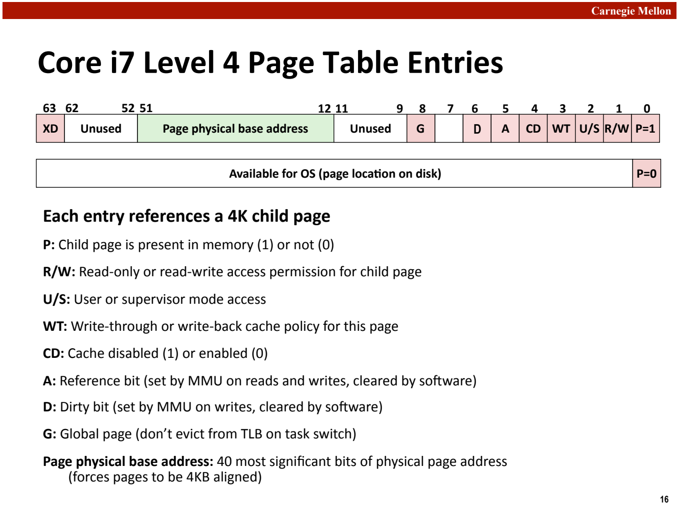
PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

15

Core i7 Page Table Entries



Carnegie Mellon

Core i7 Level 4 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1

Available for OS (page location on disk)	P=0

Each entry references a 4K child page

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

CD: Cache disabled (1) or enabled (0)

A: Reference bit (set by MMU on reads and writes, cleared by software)

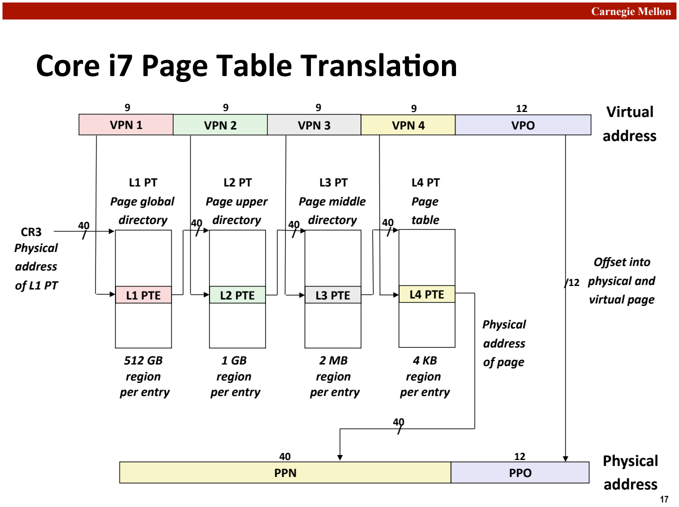
D: Dirty bit (set by MMU on writes, cleared by software)

G: Global page (don't evict from TLB on task switch)

Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

16

Core i7 Page Table Translation



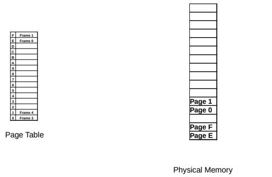
Carnegie Mellon

Core i7 Page Table Translation

17

Remembering paging

- Addresses spaces are not necessarily contiguous. Lets think of a HYPOTHETICAL 16 bit system with 4K pages (OVERSIMPLIED EXAMPLE)
  - lets think of a process with a small address space (like the one



ONE LEVEL PAGE TABLE FOR PROCESS WITH A SMALL ADDRESS SPACE

previously seen)

Remembering paging

- If the process were to reference address 0x12ff, the MMU would translate that reference to 0x42ff PHYSICAL memory address (as we can see in the figure that page 1 of the process is at frame 4)
- But what happens if the proccess were to reference address 0x542a?
  - We can see from the figure that the proccess DOES NOT HAVE page 5 in its address space.
  - The MMU would produce an exception and transfer the control to the Operating System
  - The Operating System would have the proccess killed because it has referenced and invalid address
- How does the MMU know that that is not a valid address??
  - The page table entry has one bit (typically called presence bit) that states whether the page is in memory or not

Taking advantage of paging

- Now we can take advantage of the working of the paging system:
    - We do not load a process completely into memory
    - The pages not loaded are marked with their presence bit to 0 (as are the pages not part of its address space)
    - When a page with its presence bit 0 is referenced, an exception is produced and control goes the operating system

Taking advantage of paging

- When the O.S. gets control, called as the result of a MMU exception it checks why the exception has happened
  - The referenced page is in fact an invalid address? Then the O.S. has the process killed (as seen before)
  - The referenced page is a valid address that the O.S. did not load when loading the process? The the S.O. loads it now, updates the page table and returns control to the process
- The process retries the last instruction: its state was saved when control was transfered to the O.S. as the result of the exception
- Now the page is IN MEMORY (presence bit 1) and execution continues normally
- This is the most common implementation of VM, called *demand paging*

Where are the pages?

- This poses another question. If a page is not in memory..where is it? Where does the operating system take it from?
- That deppends on whether it is a page of code, data or stack and whether it has been modified

code Code pages, which are read only, are taken from the executable file

data New (unmodified) data pages are taken from the executable file when the data in them have been given initial values. If the data they contain do not have initial values then those pages are newlly allocated and zeroed (explaining why in C external and static variables without explicit initialization are in fact initialized to 0). Already used (modified) data pages are taken from the *swap* device

stack New (unmodified) stack pages are newlly allocated and zeroed. Already used (modified) stack pages are taken from the *swap* device.

- So to have virtual memory we need some storage device called *swap device* to store the modified data and stack pages

What do we need to have VM?

The most common implementation is *demand paging*. This requires

1) a paging addressing scheme with the presence bit, so that when a page is referenced and its presence bit is 0, the MMU produces an exception
2) some storage device configured to hold pages of processes, called *swap device*
   - this *swap device* can be either a file or a disk partition
   - Windows O.S.s typically use a file, unix O.S.s typically use a partition
   - Advantages of using a file: flexibility (the file size can change)
   - Advantages of using a dedicated partiition: speed (no need to use file system indirections)

What do we need to have VM?

3) an Operating System capable of dealing with the aforementioned exception in the following way
   - Save the process state
   - If the address is in fact an illegal address: take the appropiate action to deal with illegal addresses (typically killing the process)
   - If the address is a valid address but the page is not in memory
     - Locate the page in the *swap device*
     - Read the page into memory.
     - Update the page table for the process. (that very page is now IN memory)
     - Restore the process state and continue execution

VM is not that simple

- In the previous slide we made VM look quite simple but there are a lot of things to consider. Lets think of the simple sentence **“read the page into memory”**. Several considerations con be made here
  - Do we schedule another process to run in the meantime? (probably yes, as the reading implies waiting on a device)
  - Do we have a free frame?. Would it be convenient to have a pool of free frames?
  - If we do not, we’ll have to make some frame free by replacing the page contained in it. Would it be a page of the very same process? (that’s what we will call *replacement scope*)
  - How do we decide which page to replace? (that’s what we will call *replacement policy*)
  - What if the page to be replaced has been modified? Shouldn’t it be written to the *swap* first? (we call this *cleaning policy*)
  - Do we bring into memory just one page or maybe we try to anticipate the needs of the process and bring several pages (we call this *fetching policy*)

Segmentation vs paging

- we've seen what (hardware) segmentation and paging were
- we even have seen paged segmentation real examples of a (two level) paged segmentation system (intel's 32 bit PC architecture)
- sgementation, on one hand, is closer to the logical structure of a program (separate regions for code, data, heap, libraries stack . . .)
- paging, on the other hand, is much more memory efficient, as pages are all the same size. Each page can be replaced with anothe page.
- so, most O.S. implement virtual memory with demand paging (still there were some O.S. which didn't, such as IBM's OS/2 1.x, which implemented it with demand segentation)
- most present hardware provides only paging, and for hardware that provides also hardware segmentation such segmentation is rarery used (for example unix'es in the intel 32bit PC architecture do not use the hardware segmentation)

So..what is a software segment?

- Lets think of an (again!!) OVERSIMPLIFIED example
- Let us consider an 16 bit paged system : 6 bits page number, 10 bits page offset (1k pages).
- Now we consider a (really simple) process that has

code	page0, page1, page2
data	page4, page5, page6, page7, page8
stack	page38, page39

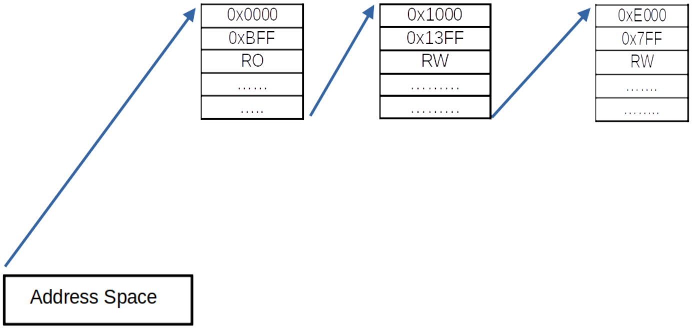
So..what is a software segment?

- If we translate that to logical address, we have that this process has

code	starts at address 0x0000, size 3k (0xC00). Ends at adress 0x0BFF
data	starts at address 0x1000, size 5k (0x1400). Ends at address 0x23FF
stack	starts at address 0xE000, size 2k (0x800). Ends at address 0x 0xE7FF

- Let's now imagine that the O.S. keeps the address space of the process, as a list of items, describing the different *"zones"* of the address space
- Each item has the starting virtual address, the size, the permissions . . .

Address space of the process as a list



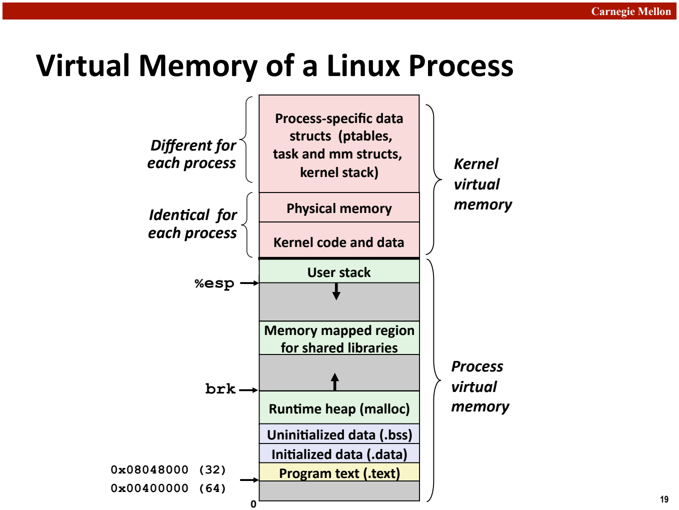
Page faults and segmentation faults

- Lets now imagine that all of the process pages, execpt page5 and page6 are in memory. This means that presence bits for page0, page1, page2, page4, page7, page8, page38 and page39 are 1. The rest of the presence bits are 0.
- If the process now references address 0x124A, which is in page4, the MMU would translate that to the correspondig real address
- If the process references address 0x14f2, which is in page5, as the presence bit is 0, the MMU would produce and execption. The O.S., upon being given control would check that that address is IN the range 0x1000-0x23ff, so its a valid address. This is a page fault, and the O.S. will serve the page fault.

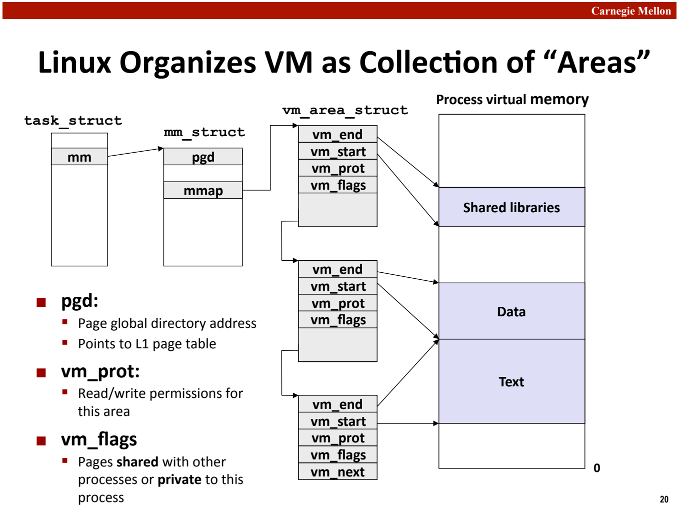
Page faults and segmentation faults

- If the process were to reference address 0x4f06, which is at page 0x13 (19), as the presence bit is 0, the MMU would produce and execption. The O.S., upon being given control would find that that address is not in any of the segments. That is in fact an invalid address, proddcing a *segmentation fault* kind of error. typically resulting in the process being killed.
- We can see that we can have *segments* when there is not hardware segmentation
- Linux calls these entities *segments*, while other O.S. call them *regions*. The command *pmap* shows the segments for a process (with their starting address, size and permissions)

Memory map for a process in Linux


Carnegie Mellon
Virtual Memory of a Linux Process
Process-specific data structs (ptables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Different for each process
Identical for each process
Kernel virtual memory
User stack
%esp
Memory mapped region for shared libraries
brk
Runtime heap (malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
Process virtual memory
0x08048000 (32)
0x00400000 (64)
0
19

Address space of the process as a list



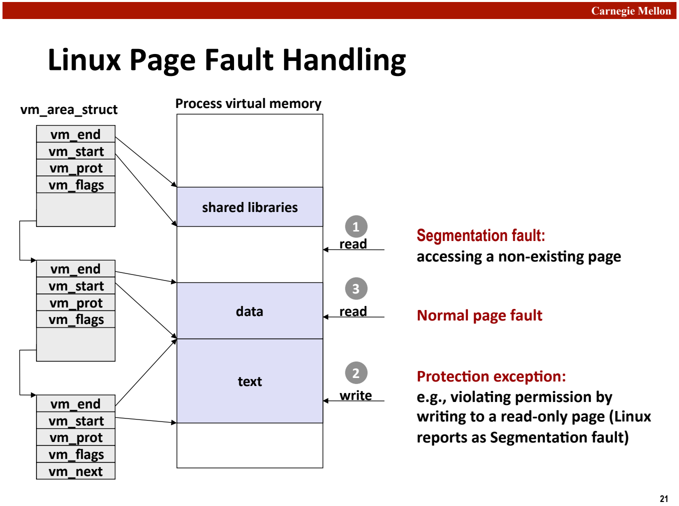
Carnegie Mellon

Linux Organizes VM as Collection of “Areas”

- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages **shared** with other processes or **private** to this process

20

Page fault vs Segmentation fault



Linux Page Fault Handling

vm_area_struct
Process virtual memory
vm_end
vm_start
vm_prot
vm_flags
shared libraries
vm_end
vm_start
vm_prot
vm_flags
data
text
vm_end
vm_start
vm_prot
vm_flags
vm_next
1
read
3
read
2
write

Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

21

Types of segments

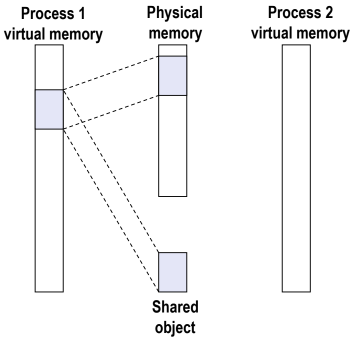
- There are two types of segments
  - **vnode segments**. They are segments associated to a file, such as the code segments, mapped files. . . . Pages from these segments get initial values from the file they are associated to.
  - **anonymous segments**. They are not associated to a file. Examples of this are the data, and stack segments. They are ultimately associated to the *swap device*. Pages from these segments get initial values of 0s. (They are zeroed when they first fault)
- In linux (and Solaris) the command `pmap` shows the segments in the address space of a process, together with their starting address and size, and in the case of a *vnode segment*, the file it is associated to
- In FreeBSD the command `procstat vm` shows the segments in the address space of a process, together with they starting and ending virtual addresses, and in the case of an *vnode segment* the file it is associated to

Copy on write

- There are situations when **segments need to be duplicated** (for example during the *fork()* system call)
- If the segments are readonly they are simply shared
- If the segments are writable, the following procedure is applied (called *copy-on-write*)
  - The segment is shared and **marked read only**
  - An attempt to write to that segment will result in an exception
  - As a consequence of the exception control goes to the O.S.
  - Upon checking that the exception is produced by trying to modified a originally marked writable segment now readonly, the O.S. will duplicate the page producing the exception, and update the page composition for one of the segments.
  - This avoids dupplicating the whole segments. Only the pages actually modified are duplicated.

Duplicating a Segment: original mapping

Sharing Revisited: Shared Objects

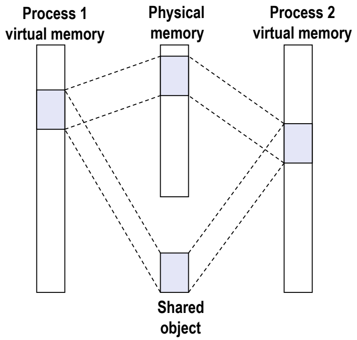


- Process 1 maps the shared object.

Duplicating a Segment: sharing the segment

Carnegie Mellon

Sharing Revisited: Shared Objects

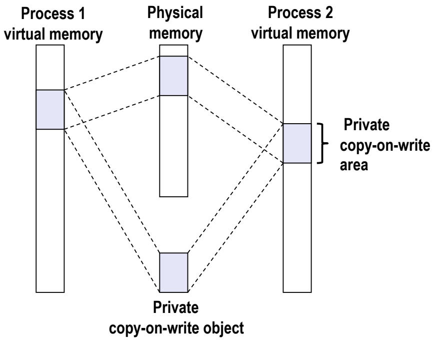


- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.

26

Duplicating a Segment: copy on write

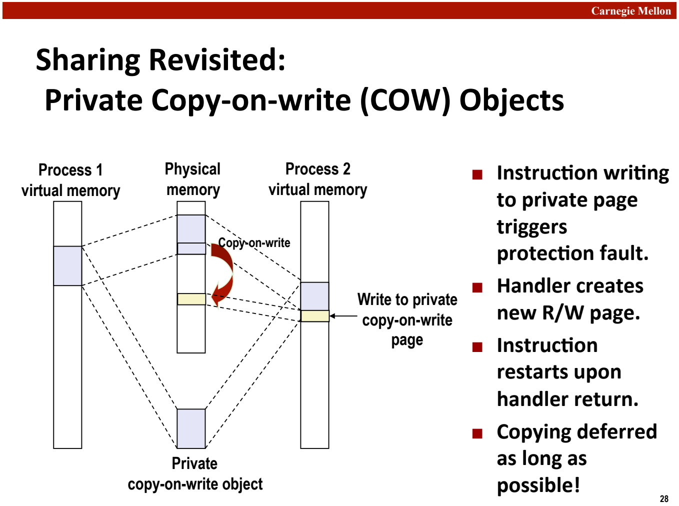
Sharing Revisited: Private Copy-on-write (COW) Objects



- Two processes mapping a ***private copy-on-write (COW)*** object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

27

Duplicating a Segment: COW, duplicating modified page



Sharing Revisited: Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

28

What is Virtual Memory exactly and why does it work?

Virtual Memory

- VM is the ability to execute programs that are not completely loaded into memory
- It makes sense because some portions of the program may be not used at all
  - very rare error conditions
  - options that never get used
  - oversized variables

Advantages of virtual Memory

- Having the ability to execute a program that is not completely loaded has these advantages
  - Progtams can be bigger than the physical RAM installed on the machine
  - The degree of multiprogramming can be increased
  - Swapping programs uses less i/o
- The most usual way of implementing VM is with *demand paging*
- It can be implemented with *demand segmentation* but it is not as efficient as compactations need to be made (IBM OS/2 v1.3)

Demand paging

- We saw previously the operation of demand paging
- Programs larger than physical memory can be executed at the cost of a reduction in speed
- Since pages often have to be brought into memory, the problem of which page to replace arises.
- In addition to the different algorithms, it is necessary to consider whether local or global replacement is used.
- Another aspect to take into account is the criteria for assigning frames to the processes

Performance of demand paging

- The page fault service includes the following tasks
  a) Serve page fault exception
     - Transfer control to the O.S.
     - Save process state
     - Determine that the exception is a page fault
     - Locate the page in the *swap device*
     - Allocate a new frame
  b) Transfer the page
     - Wait in device queue
     - Wait for seek and latency times
     - Transfer page into frame
     - Update page table
  c) Resume process
     - Restore process state
     - Resume execution

Demand paging performance: example

- Let's assume a system that operates at 2GHz, where the memory access time is in the order of 5 ns (typical value of a DDR2 memory), and where the swap device has a average search time of 9.5 ms and a transfer speed of 40Mb/second
- In this case we can estimate an upper bound of 10ms to serve the page fault (9.5ms for the seek time, 0.1 ms for the transfer of a 4K page and an additional 0.4ms which are more than enough for latency and the rest of the tasks.
- If the probability of a page fault is 10^{-6} (one page fault each million of memory refferences), the mean effective access time would be $t.a.e. = (1 - 10^{-6})x5ns + 10^{-6}10ms \approx 5ns + 10ns = 15ns$

Demand paging performance: considerations

- Increasing the physical memory of the machine decreases the probability of a page fault
- As the probability of a page fault decreases, the effective access time decreases and therefore the apparent execution speed increases.
- Virtual memory allows running a process that does not fully reside in memory at the cost of a decrease in execution speed
- In order for a system with virtual memory to function optimally, the number of page faults must be minimal

What is Virtual Memory exactly and why does it work?

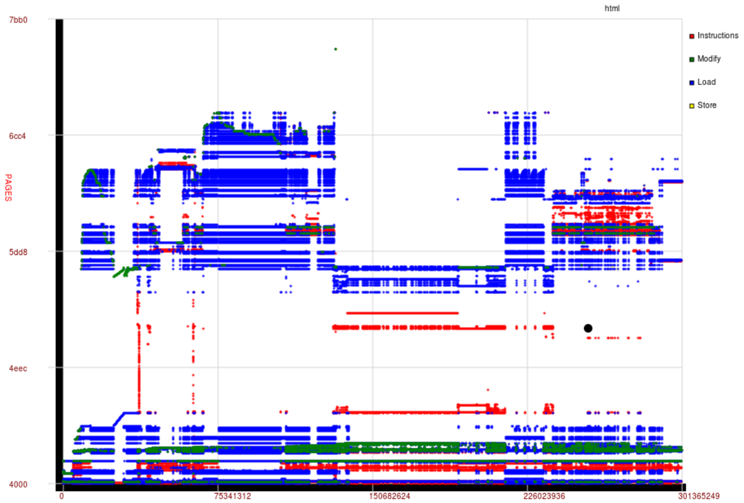
Principle of Locality

- This principle states that as a program executes it moves from one location to another
- A location is a set of pages that are referenced actively and together
- A program in general is made up of several locations, which can overlap
- In other words: the sets of pages referenced by a process during its execution are grouped in space (*spatial locality*) and in time (*temporal locality*)

Spatial and temporal location

- **Temporary location:** Recently referenced memory addresses are likely to be referenced again in the near future
  - loops
  - functions, procedures, subroutines . . .
  - stacks
  - counters, accumulators. . .
- **Spatial location:** If a memory address is referenced it is likely that a nearby location is referenced
  - arrays and structures
  - sequential execution
  - related variables are often declared together

Locality of a program



How does principle of locality benefit VM?

- The program goes through differente localities (depending on which stage of its execution it is)
- The O.S. must adapt its Resident Set
- If the Resident Set is equal to the locality of the process, no page faulst will happen until the next change in locality
- Page faults will increase when the program is switching localities

Example of various localities in a program

- Consider the oversimplified example in the next page.
- We have declared the variables as external, so they are stored in the data segment, and we can **limit our reasoning to pages in the data segment**, making it clearer.
- Asuming 4 byte floats and a page size of 4k (reasonable assumptions), each row of matrixes a, b and c occupies 8 pages (matrixes in C are stored by rows). Each of those matrixes uses up to 256 Mb (64 K pages).
- If we allocate 3x64 K pages to our process data , we get one fault to load each page and no more page faults one they are loaded. Process data would use 192 K pages, which we will have allocate to the process all of its execution (remember, in this oversimplified example we only consider the data pages, nor the code neither the stack pages)

Sample program with diferent localities

```
#define MAX 8192
float a[MAX][MAX],b[MAX[MAX],c[MAX]MAX];
......
void MultiplyMatrix (float x[MAX][MAX], float y[MAX][MAX], float z[MAX]MAX])
{
   int i, j, k;

   for (i=0; i<MAX; i++)
      for (j=0; j<MAX; j++){
          x[i][j]=0;
          for (k=0; k<MAX; k++)
             x[i][j]=x[i][j]+y[i][k]*z[k][j];
        }
}

main()
{
   ReadMatrix(a); /*locality one*/
   ReadMatrix(b); /*locality two*/
   MultiplyMatrix(c,a,b); /*locality three*/
   PrintMatrix(c); /*locality four*)
}
```

Example of several localities

- The process goes, however, through four different localities.
- During localities one, two and four the process acesses only one matrix at a time. If fact, as it acceses it secuentially by rows, allocating ONLY ONE frame to the process during that locality will produce as many page faults as allocating 192K frames (remember we are considering only data pages)
- During locality three , the process would need (again considering only data pages): One page for the element of matrix c (parameter x), 8 pages for the row of matrix a (parameter y) and 8192 pages for the column of matrix b (parameter z).
- The progam can do with (oversimplified example, plus considering only data pages)
    - Locality one: one frame, 64 K page faults
    - Locality two: one frame, 64 K page faults
    - Locality three: 8201 frame 192K page faults
    - Locality four: one frame, 64 K page faults

What is Virtual Memory exactly and why does it work?

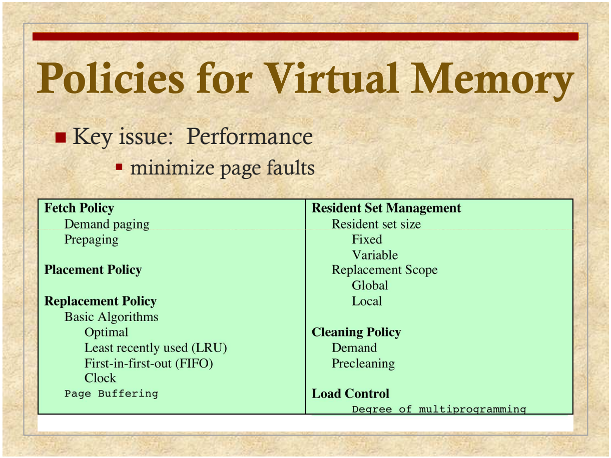
Thrashing

- *Thrashing* occurs when a system spends more time paging than executing
- A process that has fewer frames allocated to it than it is actively using, will continuously produce page faults since each failure will replace a page which is also used
- If the replacement is global (one process can replace pages of others) it can spread the problem two other processes
- To avoid *thrashing*, an attempt is made to assign a sufficient number of frames for each process to execute smoothly. As we will see later , there are two ways for the O.S. to determine the correct number of frames to allocate to a process
    - the *working set* model
    - *page fault frequency* model

Virtual memory considerations

- When designing a Virtual Memory system, there are a lot of things to take into acount, each with its own considerations
  - Where in memory is the newly referenced page going to be placed?: **page placement policy**
  - Is the page causing the page fault the only one brought into memory?: **page fetching policy**
  - Is the allocation of frames to a process fixed or variable in time?: **frame allocation policy**
  - Does the page causing the fault replace one of the very same process?: **replacement scope**
  - How does the page to be replaced get chosen?: **replacement algorithms**

Virtual memory considerations



Policies for Virtual Memory

- Key issue: Performance
  - minimize page faults

Fetch Policy Demand paging Prepaging **Placement Policy** **Replacement Policy** Basic Algorithms Optimal Least recently used (LRU) First-in-first-out (FIFO) Clock Page Buffering	**Resident Set Management** Resident set size Fixed Variable Replacement Scope Global Local **Cleaning Policy** Demand Precleaning **Load Control** Degree of multiprogramming

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Page placement

- With *page placement policy* we refer to the decission of where to put the new page in memory
- This is not a concern in paging systems as all the memory frames are equal.
- It world be a concern in segmented system
- Not a concern in a paged segmentation system as the memory is ultimately allocated in pages
- It would be a design issue in NUMA (Non Uniform Memory Access) systems

Page fetching

- When a page produces a page fault a choice can be made
  - bring ONLY that page into memory: **pure demand paging**. (this also implies that the O.S. does not load any page of a process until it is referenced)
  - bring that page and some others into memory: **prefetching**
- *prefetching* exploits the characteristics of most *swap devices*: it is faster to bring four pages at one time than bringing four pages one at a time. There's no warranty that all the pages will be referenced: it might be inefficient
- *pure demand paging* produces a lot of page faults when the process is started, although as more pages are brought in, the probability of a page fault decreases

Page replacement

- In the management of virtual memory, what is called *replacement policy* takes great importance. It is what decides which page of those present in memory is the one that goes to be replaced
- In this *policy* several concepts are involved, which although different, are strongly interrelated
  a) The amount of physical memory (number of frames) allocated to each process on the system (*frame allocation*)
  b) If the set of pages to be taken into account when being replaced includes only those of the process that caused the page fault or all those residing in memory of the different processes (*replacement scope*)
  c) Among the pages considered, which one should be selected to be replaced (*replacement algorithm*)

Frame assignment and page replacement

- Point *a)* raises what is known as the *frame assignment problem*. The solutions to this problem are to try to assign to each process a variable number of frames that adapts to the locality of the process. (that is, it adapts to the process' memory needs in the different phases of its execution)
- The point *b)* raises what is known as *local replacement* or *global replacement*
- The point *c)* is what is usually referred to as *replacement policy* or *page replacement algorithms*

Frame allocation

- When allocating frames for the execution of a process there are two approaches
  - **Fixed allocation**: The number of frames allocated to a process is fixed
  - **Variable allocation**: The number of frames allocated to a process evolves in time with the memory needs of the process
- The case of fixed allocation rises the question of how can the amount of frames allocated to a process be determined
  - The same for all processes
  - Proportional to the process size
  - Other considerations: process priority, cpu time. . .

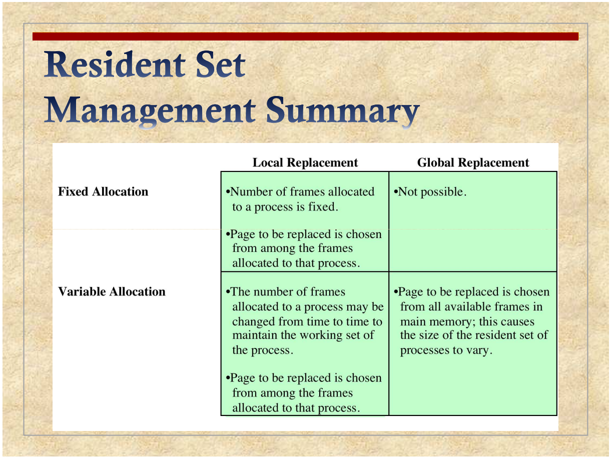
Variable frame allocation

- In the case of variable frame allocation the O.S. must guess the size of the optimal allocation, the one that better adapts to the process needs at each time.
- If the allocation is smaller than the actual needs of the process, it will produce a lot of page faults
- If the allocation is too large: memory is wasted.
- We'll see two methods of determining the needs of the process
    - The working set model
    - The page fault frequency algorithm

Replacement scope

- As replacement scope is concerned, two approaches can be thought of
    - Local Replacement: A page fault of a process can only replace a page of that process
    - Global Replacement: A page fault from one process can replace a page from another process
- Not all combinations are possible: for example there can be no global replacement with a fixed frame allocation technique

Virtual memory considerations



Resident Set Management Summary

	Local Replacement	Global Replacement
Fixed Allocation	•Number of frames allocated to a process is fixed. •Page to be replaced is chosen from among the frames allocated to that process.	•Not possible.
Variable Allocation	•The number of frames allocated to a process may be changed from time to time to maintain the working set of the process. •Page to be replaced is chosen from among the frames allocated to that process.	•Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Replacement and assignment

- At present, most Operating Systems (linux, BSD . . .), use variable assignment with global replacement
  - Very easy to implement
  - Combines with the existence of *"pool"* of free frames

Page cache: *pool* of free frames

- Most systems have a *pool*) of free frames
- If a replacement needs to be made
    - Bring the new page to one of the free frames in the *pool*)
    - The *replaced* page is now marked as free and added to the "*pool*" of free frames
- No waitting is needed to bring the new page into memory
- In the event of a new page fault, it is possible that the page that caused this new failure is in memory in the *pool* of free frames and there is no need to bring it from disk
- This technique is called *page buffering* and we call this *pool* the *page cache*

Page cache: *pool* of free frames

- As as extension to the page cache seen before, pages *stealed* from the processes, are classified in to lists
  - Free page list. Pages stealed from the processes, that are replaceable.
  - Modified page list. Pages *stealed* from the processes. As they are modified they are not directly replaceable. They are writen in clusters to the swap device as the workload of the paging device allows. After they are written to disk the go to the *Free Page List*
- When one page is referenced the O.S. checks whether it is already in the page cache, in which case it gets used and the page fault is extemely fast (no need to read from disk)
- If the page producing the page fault is not in the page cache, one of the pages in the cache gets replaced

Frame locking

- The O.S. can lock frames in memory by associating that frame a *lock* bit
- By frame locking we reffer to the ability to protect the page stored in that frame from being replaced
- Pages involved in i/o, and key structures are locked in memory. The kernel is usually in locked pages too
- As all i/o is done through system buffers, this is usually not a concern for process pages

Replacement algorithms

- Whether we have local or global replacement, we still have a decission to make: once a page fault happens whch one of the resident pages gets replaced
- This choice is done by what we call the *replacement algorithm*
- The same algorithms can apply to both local and global replacement, and they work exactly the same, considering
  - In a local replacement scope only the pages of the faulting process are considered
  - In a global replacement scope ALL the processes pages are considered

Classical replacement algorithms

- The classical page replacement algorithms are
  - **óptimal**
  - **LRU**
  - **FIFO**
- As we will see, only the FIFO algorithm can be implemented. The Optimal cannot be implemented and the LRU would need special hardware to be implemented efficiently

Other replacement algorithms

- The most usual thing is that the hardware provides a reference bit and a modified one (*dirty bit*) for each page. With this help, the most common algorithms are
  - FIFO with second chance (sometimes refered as clock algorithm)
  - Not Used Recently
  - Neither Used nor Recently Modified
  - Other LRU approximations . . .

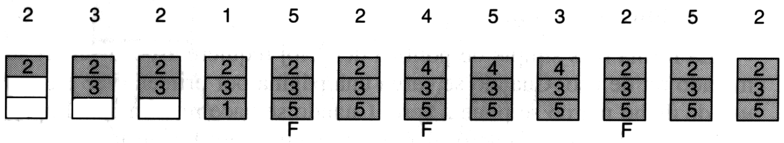
Page Replacement Algorithms

Óptimal Algorithm

- It is the one that produces the fewest page faults for any number of frames
- The page that will take the longest to be referenced again is the one replaced
- It cannot be implemented as it would imply knowing in advance which pages will be referenced in the future
- Although it cannot be implemented, it is used as a comparison reference for the other algorithms

Optimal algorithm: example

- Reference chain *2 3 2 1 5 2 4 5 3 2 5 2* with three frames produces 3 failures



Page Replacement Algorithms

LRU algorithm

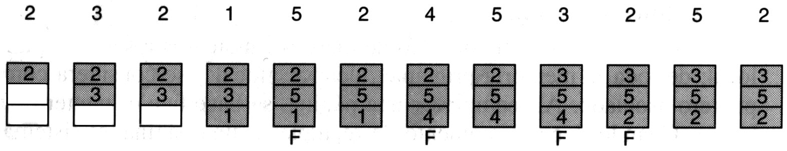
- Replaces the least recently used page
- It is like the optimum but with the chain of references reversed in time
- Produces good results due to the principle of temporal locality, recently referenced pages are likely to be referenced soon
- It adapts very well to the location of the program

LRU algorithm

- Two possible implementations
  - **Counters** A counter that represents the moment in which it was referenced is associated with each page. The one that has not been referenced for the longest time can be determined by the value of the counter
  - **List** Each referenced page is placed at the end of a list, the first in the list is the one that has not been referenced for the longest time
- None of the implementations is feasible due to the burden that would be involved in managing the counters (or the list) **with each memory reference**

LRU algorithm: example

- Reference chain *2 3 2 1 5 2 4 5 3 2 5 2* with three frames produces 4 failures



LRU approximations

- The LRU algorithm produces reasonably good results but cannot be implemented.
- On systems with on-demand paging the *hardware* typically provides a *reference bit* and a *dirty bit*
- With these aids, algorithms that approximate the LRU are usually implemented with the idea of adapting to the locality of the process
- The most common are *second chance* (clock), use of additional reference bits . . .

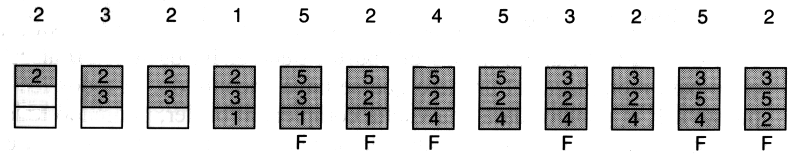
Page Replacement Algorithms

FIFO algorithm

- The first page to enter is replaced (FIRST IN FIRST OUT)
- Very simple implementation.
- It suffers from the *Belady* anomaly: for certain specific examples of memory reference chains it is possible that increasing the number of frames increases the number of page faults
    - Example: test the reference chain *1 2 3 4 1 2 5 1 2 3 4 5* for three and four frames

FIFO algorithm: example

- Reference chain *2 3 2 1 5 2 4 5 3 2 5 2* with three frames produces 5 failures



Belady anomaly

- In the previous algorithm (FIFO) we saw that for specific strings of reference, increasing the number of allocated frames might increase the number of page faults
- This is called the Belady anomaly
- *stack algorithms* do not suffer from Belady anomaly
- A replacement algorithm is said to be a *stack algorithm* if the resident set with N frames allocated is a subset of the resident set with N+1 frames allocated

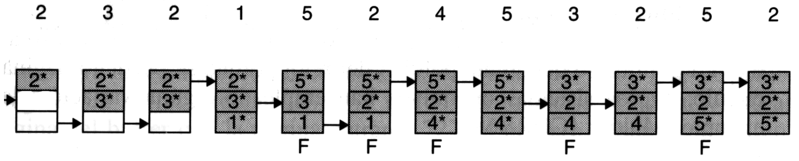
Page Replacement Algorithms

Second Chance Algorithm

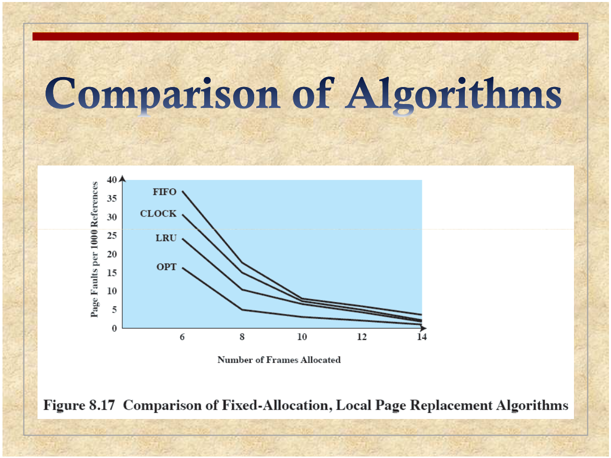
- It is basically a FIFO in which the reference bit is also taken into account
- Very simple implementation: circular queue with the pages in which the reference bit is also used
- When you have to replace, look at the índex that indicates the next page to replace
    - If the reference bit is at 0, it is replaced
    - If the reference bit is at 1, it is set to 0 and the índex is advanced to the next (the page gets a *second chance*). . .
- It is often refered to as the *clock* algorithm as the index going through the circular queue *"resembles"* a clock

Second Chance algorithm: example

- Reference chain *2 3 2 1 5 2 4 5 3 2 5 2* with three frames produces 5 failures
- The symbol $*$ represents the reference bit set



Algorithhm comparison



Comparison of Algorithms

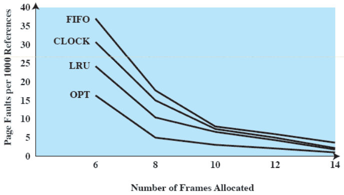
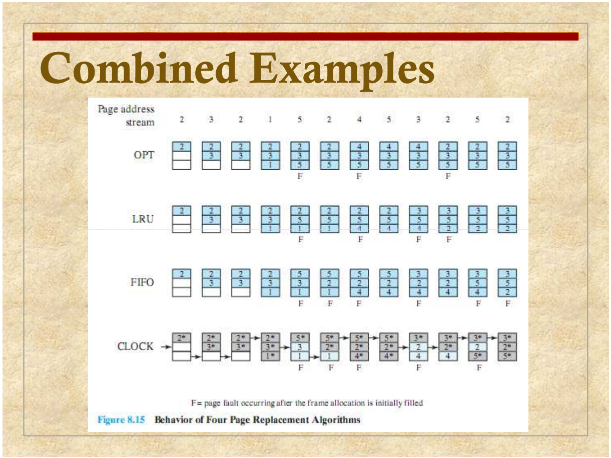


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Alhgorithm comparison



Combined Examples

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

Figure 8.15 Behavior of Four Page Replacement Algorithms

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

Page Replacement Algorithms

Variations of the Second Chance algorithm

- Using the circular page queue but with the (reference,modification) bit pair
  1. From the current index the first page is searched with (0,0) and replaced
  2. If none are found, the queue is traversed again, looking for a page with (0,1). The first found is the one that is replaced. At the same time that the search is being carried out, the reference bits of the pages through which it is passed are cleaned
  3. If none are found, go back to step 1

Variations of the Second Chance algorithm

- Using two índices to traverse the list.
  - Pages that are examined with the first index have the reference bit cleared.
  - The pages that when examined with the second index have the reference bit set to 0 are marked as free
  - The algorithm is executed at regular time intervals (depending on the amount of free memory) and the pages that are kept in memory are the ones that have been referenced from the time they were examined with the first index until they are examined with the second index
  - The page stealing process in UNIX uses this algorithm, also called the *two handed clock algorithm*

Use of additional reference bits

- The OS saves a counter for each page
- The OS periodically it checks the reference bits of each page, and saves it in its counter, introducing it to the left and shifting the other bits to the right
- The page with the smallest counter value is an approximation of the *least recently used*
- Is not **actually** the **least recently used** because bits are checked at certain time intervals, not every memory access
- For example, if we have 3 pages A, B and C whose counters are 1110000000000000, 1110000000000000, 0000000001000000, respectively,
  - The page that has not been referenced for the longest time is C (the last 8 times that the OS checked the bits, it had not been referenced
  - Pages A AND B have both been referenced during the last three time intervals in which S.O. checked the bits, however we don't know which one has been referenced more recently

Scope and allocation combined

- As we have seen, attending to the scope, replacement can be
  - Global replacement
  - Local Replacement
- And as for the allocation of frames to a process we have
  - Fixed allocation
  - Variable allocation
- This yields, in principle, four scenarios
  a) Global replacement with fixed allocation
  b) Global replacement with variable allocation
  c) Local Replacement with fixed allocation
  d) Local Replacememt with variable allocation
- Option *a)* is clearly not possible. The others have their own particularities

Global replacement with variable allocation

- It is the approach adopted on modern operating systems (Linux, FreeBSD, Solaris . . .)
- Quite easy to implement
- In theory, when a process produces a page fault a new frame is added to the resident set of that process. The page to be replaced is selected from all the pages in the system using one of the aforementioned algorithms.
- Modern operating systems work with a *pool* of free pages and the new page is simply added to the resdent set of the process. On the other hand the *page stealer* process tries to free (*steal*) pages from the processes, typically using the two handed clock algorithm

Local Replacement with fixed allocation

- In this approach, the Resident Set Size of a process remains the same through its execution
- Not used in modern systems
- It makes necessary to decide the amount of allocation to give a process when the process is loaded
  - It it is too small, the process will produce a lot of page faults
  - If it is too large, the system is wasting memory and probably under-multiprogrammed

Local Replacement with variable allocation

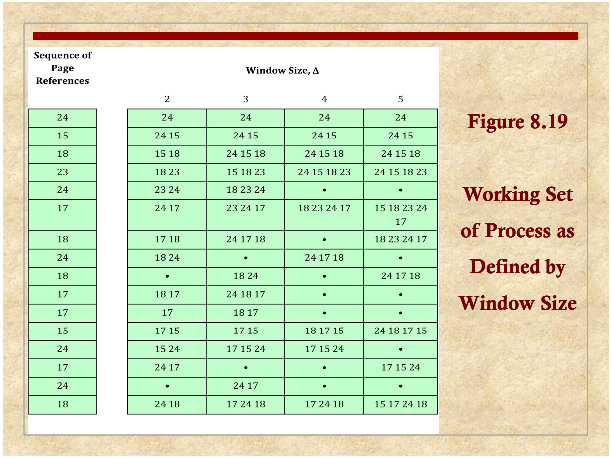
- The process is allocated a number of frames
- When it produces a page fault, one of the pages is replaced (with one of the algorithms seen)
- The system tries to determine whether the Resident Set Size is adecuate for the process execution
    - If it is too small (many page faults) the system will allocate more frames to the process
    - If it is too large, the system will try to allocate less frames to the process
- How does the system find out whether the Resident Set Size is adecuate for the process execution?. Two solutions
    - Working Set model
    - Page Fault Frequency algorithm

Adapting the Resident Set Size

Working Set

- It is based on the principle of *locality*
- For a process, the *working set* of window Δ at an instant t, $W(t, \Delta)$ is defined as the set of pages referenced between instants $(t - \Delta)$ and t
- These are the pages referenced in the last Δ moments
- If Δ is chosen appropriately it can be adapted to the locality of the program. If Δ is too small, it will not cover the entire locality; if Δ is too large, it may overlap several localities
- The idea is to assign each process the necessary number of frames so that it can contain its working set

Working Set with different windows sizes



Sequence of Page References	Window Size, Δ			
	2	3	4	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

Figure 8.19

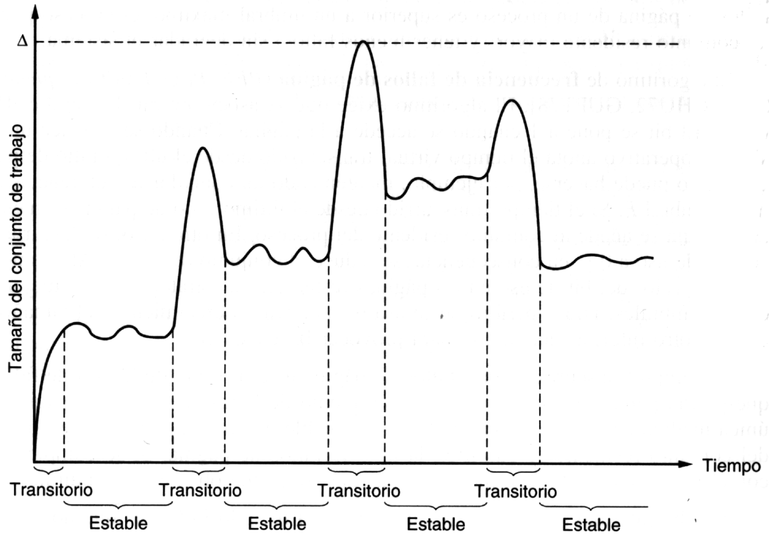
Working Set of Process as Defined by Window Size

Figure: William Stallings. Operating Systems. Internals and Design Principles, seventh edition, Pearson Education 2012

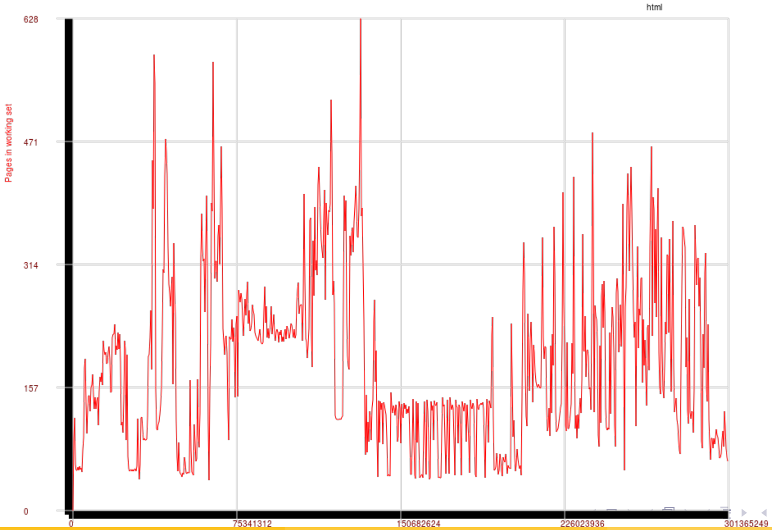
Working Set

- The Working Set of a process changes throughout its execution
- During some stages of the execution of a process it remains stable
- Between stable stages there are transitory periods
- The OS: must keep account of the WS of all processes so that the physical memory used corresponds to the sum of the WS of the running processes
    - If a process cannot cannot be allocated its Working Set, the system will suspend its execution until more memory is available

Working Set Ideal



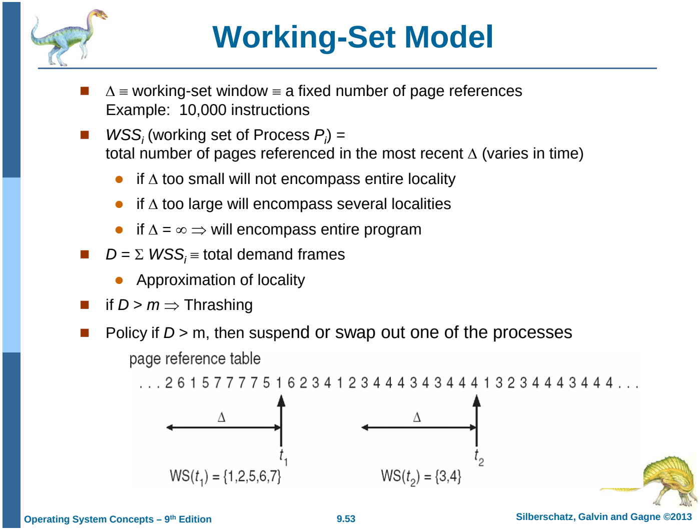
Working Set Real



Working Set

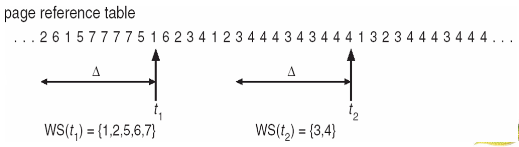
- A Working set of window Δ at an instant t, $W(t, \Delta)$ is not the same as the Resindent Set of algorithm LRU with Δ frames
  - $W(t, \Delta)$ Is the set of pages referenced in the last Δ references
  - LRU with Δ frames is the last Δ pages referenced.
- Example. Consider the following string of references `ABCDEABABABABABAB`
  - the Worrking Set with window 4 at time 15 would be { A, B}
  - a LRU algorithm with 4 frames at time 15 would yield the resident set {D, E, A, B }
- Approximations are implemented since the ideal model would involve evaluating the WS on each memory reference an thus it cannot be implemented efficiently.

Working Set Model



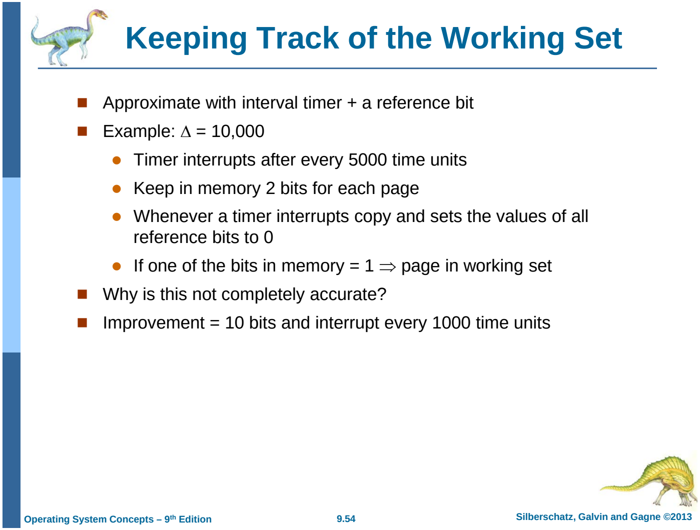
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes



Operating System Concepts – 9th Edition 9.53 Silberschatz, Galvin and Gagne ©2013

Working Set Approximation



Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Operating System Concepts – 9th Edition 9.54 Silberschatz, Galvin and Gagne ©2013

Adapting the Resident Set Size

Working Set Model
Page Fault Frequency

Page fault frequency

- It is an effective method to control thrashing
- A time (number of references) limit is established,
- When a page fault occurs, the number of references elapsed since the last fault is compared with the limit
    - If it is less than the limit, the new page is ADDED to the set of resident pages of the process (the number of frames assigned to the process is increased)
    - Otherwise, all pages that have not been referenced since the last page fault are discarded, thus decreasing the number of frames allocated to the process
- It is sufficient that the *hardware* provide a reference bit

Page fault frequency example

a	b	c	a	a	d	a	b	a	b	a	b	a	b	f	g	a	f	g

a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
		c	c	c	c	c	c	c	c	c	c	c	c	f	f	f	f	f
					d	d	d	d	d	d	d	d	d		g	g	f	g

F	F	F			F									F	F			

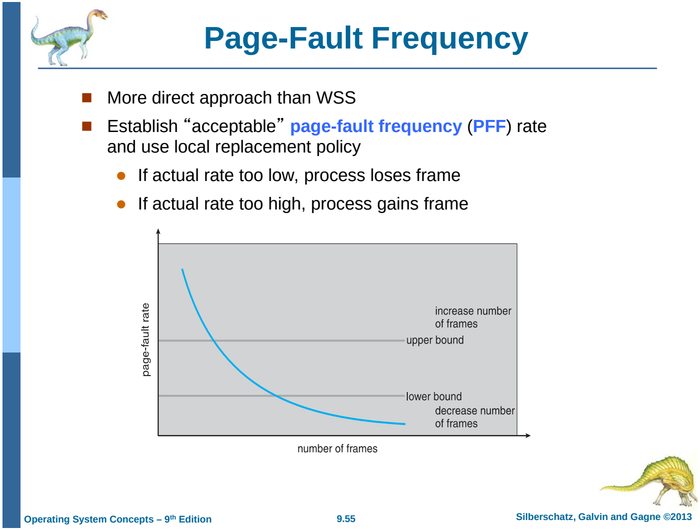
Page fault frequency example

- Let's asume the limit is 4.
- The first page faults happen in consecutime memory references (< 4), so new pages are incorporated and the resident set size increases
- The first reference to page d happents a time 5, las page fault was at time 2, $(5 - 2 = 3 < 4)$ so page d si brought to memory without taking any one out
- The first reference to page f is at time 14, as last page fault was at time 5, all pages not referenced since the las page fault are discarded, and the new page is brought in
- REMEMBER: this example is OVERSIMPLIFIED, 4 as the limit is really unrealistic, as is the number of frames involved

Page fault frequency. Alternate Implementation

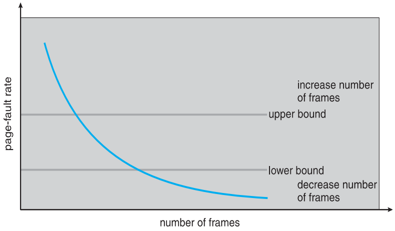
- It can also be implemented with two limits, in this case, if the number of references elapsed since the last page fault
  - is smaller than the lower limit, the new page is ADDED to the set of resident pages of the process (the number of frames assigned to the process is increased)
  - is greater than the upper limit, all pages that have not been referenced since the last page fault are discarded, thus decreasing the number of frames assigned to the process
  - is between the two limits, a page of the process is replaced

Page Fault Frequency

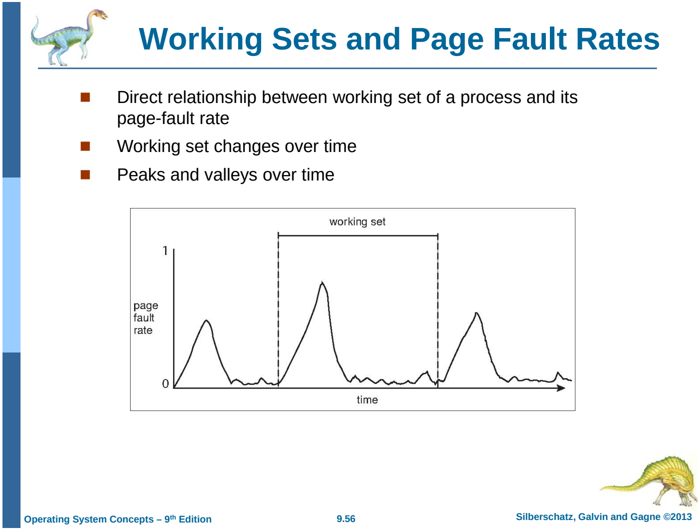


Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

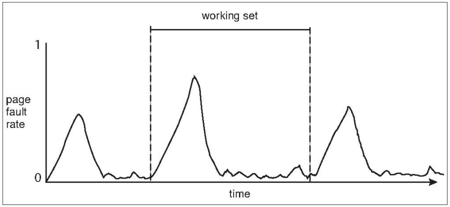


PFF and WS



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Operating System Concepts – 9th Edition 9.56 Silberschatz, Galvin and Gagne ©2013

Demand segmentation

- If paging hardware is not available, it is possible to implement virtual memory with demand segmentation
- For this it is necessary
  - A swap device
  - Segmentation hardware with presence bit that indicates whether the segment is present in memory so that:
    - A segment that is in memory is referenced: it is accessed normally
    - A segment that is not in memory is referenced: an exception is thrown
- The Intel 286 processor had segmentation hardware but not paging: IBM OS/2, until version 1.3, provided virtual memory with on-demand segmentation

Demand segmentation: replacement algorithm

- The replacement algorithms are similar to those for on-demand paging. Let's see how OS/2 did it
- The system maintained a list of segments in memory. Periodically the system
    - Placed the accessed segments at the end of the list
    - Cleared access bits
- When a segment had to be replaced, the first (or firsts, as all segmets do not have the same size) of the list was (were) replaced. This is an approximation to LRU since the list is *"sorted"* by access time

On-demand segmentation: segment replacement

- The replacement mechanism is a little different from that of on-demand paging, since the different segments have different sizes. Let's see how OS/2 did it
- When a segment fault occurs
  1. It is checked if there is enough space in memory, and if there is, a compaction is done
  2. If there is no space in memory, the first segment of the list is taken and, if necessary, written in the swap device
     - If there is enough space, the segment is loaded into memory, the segment table is updated and the segment is placed at the end of the list
     - If there is no space, return to step 1

Other considerations

- So far we have seen the following elements that influence the performance of demand paging
  - The page replacement algorithm
  - The type of replacement: local or global
  - The type of allocation of frames to a process: fixed or variable
- The page size can also have influence
  - small pages: better adapted to the locality of the program
  - large pages: simplify accounting and optimize transfers

Other considerations: data placement in memory

- Demand paging is transparent to the user and the programmer, so that, in principle, they do not have to be aware of its existence
- However, if we take into account that matrixes are stored by rows in C, the two programs shown below show very different results on the same machine due to the different way of traversing memory and therefore the different number of page faults they cause.

```
$time ./p1
actual 0m1.897s
user 0m1.236s
sys 0m0.644s
$time ./p2
actual 0m17.966s
user 0m16.905s
sys 0m0.912s
```

code of p1

```
#include <stdlib.h>

#define NCOLS 1024*16
#define NFILAS 1024*16

main()
{
  int **a;
  unsigned  i, j;

 a=(int**) malloc (NFILAS * sizeof (int*));
 for (i=0; i<NFILAS;i++)
   a[i]=(int *) malloc (NCOLS*sizeof(int));


   for (i=0; i<NFILAS; i++)
     for (j=0; j<NCOLS; j++)
        a[i][j]=23;

}
```

code of p2

```
#include <stdlib.h>

#define NCOLS 1024*16
#define NFILAS 1024*16

main()
{
  int **a;
  unsigned  i, j;

 a=(int**) malloc (NFILAS * sizeof (int*));
 for (i=0; i<NFILAS;i++)
   a[i]=(int *) malloc (NCOLS*sizeof(int));

   for (j=0; j<NCOLS; j++)
     for (i=0; i<NFILAS; i++)
        a[i][j]=23;

}
```

APPENDIX: Unix system calls related to memory management

system calls
malloc() C package
mapping files in memory
shared memory

Unix system calls to memory management

- The main system calls related to memory are
  - **brk** and **sbrk** to set the program **"break"** (end of the data segment)
  - **mmap** and **munmap** to map and unmap files in memory
  - **shmget**, **shmat**, **shmddt** and **shmctl** for shared memory management
- the **malloc** library function (and **free**) is the prefered way to allocate (or deallocate) memory, instead the **brk** and **sbrk** system calls.
- (*sbrk* is sometimes implemented as a library function)

brk() and sbrk() System calls

- Sets the end of the data segment, which is the end of the heap. Increasing the program **"break"** has the effect of allocating memory to the process; decreasing the break deallocates memory.
- *brk()* sets the end of the data segment to the addr specified as the argument, and returns 0 on success.
- *sbrk()* C function. Increments the program's data space by increment bytes. Calling *sbrk()* with an increment of 0 can be used to find the current location of the program **"break"**. On success, *sbrk()* returns the previous program break. (If the **"break"** was increased, then this value is a pointer to the start of the newly allocated memory)

APPENDIX: Unix system calls related to memory management

C malloc() package

- Allows manual memory management for dynamic memory allocation via a group of library functions.
- The library functions are responsible for heap management.
- Package for explicit assignment and releasing memory.
- In C, **the programmer is responsible for the porgram's memory management**: There are not Garbage Collectors or other aids

malloc() C package

- *malloc()* allocates the requested bytes of memory and returns a pointer to it. Writing more bytes than allocated has an undefined behaviour (*segmentation fault, heap corruption . . .*)
- *malloc()* has granularity (16, 32 . . . depending on the implementation) which means that if you allocate 3 bytes with *malloc()* you probably are being actually allocated at least 8 or 16
- *free(ptr)* releases the **memory allocated with malloc()**. *ptr* **must be an address** obtained with *malloc*.
- Should *ptr* not be an address allocated with malloc (calloc, realloc . . .), the behaviour is undefined.

malloc() C package

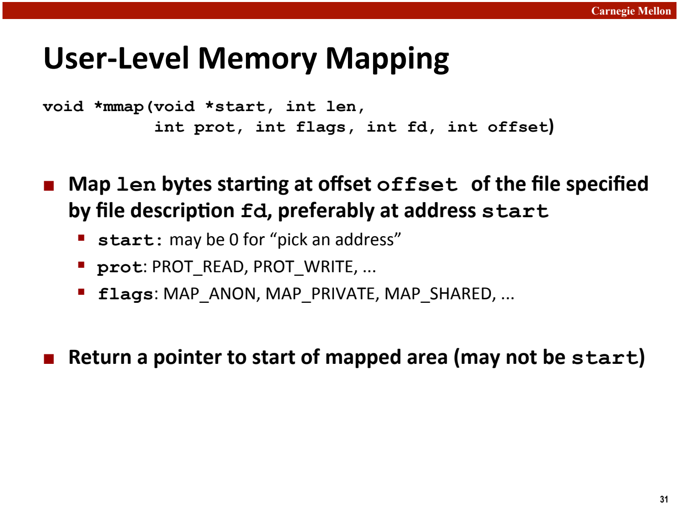
- *calloc()* assigns an initializes (to 0's)memory for n elements of size bytes each, realloc resizes a block of allocated memory
- These functions invoke the syscalls *brk()* and *sbrk()* to manage the heap.
- the *mmap()* system call maps a file into a process address space.
- In present versions of linux, allocating less than 128 Kbytes with malloc grows the heap segment. Allocating a bigger amount than 128K creates a new segment of anonymous (non associated to a file) memory in the process virtual address space

APPENDIX: Unix system calls related to memory management

mmap system call

- **mmap** maps a file into memory
- the contents of the file *appear* to be in memory starting at some address
- The mapping need not be complete neither need it be from the start of the file
- Protection (read, write . . .) also applies
- **mmap** creates a new *"segment"* in the process virtual address space

the mmap system call



Carnegie Mellon

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- **Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`**
  - **`start`:** may be 0 for “pick an address”
  - **`prot`**: PROT_READ, PROT_WRITE, ...
  - **`flags`**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- **Return a pointer to start of mapped area (may not be `start`)**

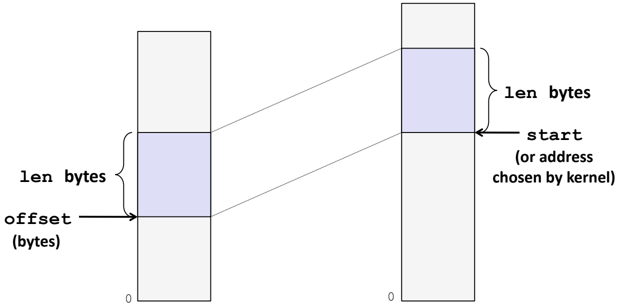
31

mapping a file in memory

Carnegie Mellon

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Disk file specified by file descriptor **fd**

Process virtual memory

32

APPENDIX: Unix system calls related to memory management

Creating shared memory

Shared memory is part of the IPC (Inter Process Communication) resources. As IPC resources are shared by several processes, it becomes necessary that different processes can refer to the same resource: every IPC resource in the system is identified by a number (its key).

1) First it is necessary to get a memory block (creating it or using one already created)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

- **key**: number identifying the resource on the system
- **size**: size of the shared memory region (must not be greater than the actual size if the segment already exists)
- **shmflg**: Bitwise OR of the permissions and one or more flags

flags on IPC resources *get* system calls

- Available flags (to be bitwise OR'ed with the permissions) are:
  - `IPC_CREAT`
  - `IPC_EXCL`
- Used as follows
  - **0** If the resource already exists, shmget will return a valid identifier, otherwise error is returned.
  - **IPC_CREAT** If the resource already exists, shmget will return a valid identifier, otherwise the resource is created and an identifier for the created resource is returned.
  - **IPC_CREAT | IPC_EXCL** If the resource already exists an error is returned, otherwise the resource is created and an identifier for it is returned.

Accessing shared memory

2) Once created, to be accessible, shared memory must be placed in the process's address space, and then it can be accessed as normal memory. *shmat "attaches"* the shared memory segment to the process address space

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int flg);
```

- **shmid**: id returned by *shmget()*
- **shmaddr**: virtual address to place the shared memory segment onto (NULL to get it assigned by the system)
- **flg**: usually 0, but can be a bitwise OR of the following: `SHM_RND`, `IPC_RDONLY`, `SHM_SHARE_MMU` (Solaris) `SHM_REMAP` (linux)

- *shmat()* returns the virtual address where the shared memory can be accessed

Accessing shared memory

3) when it is no longer needed shared memory can be detached from the process address space

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

- **shmdt** detaches the block of shared memory at the address supplied from the process address space
- If the shared memory is *"attached"* several times to the process address space(which is a possibility), it is only detached from the address supplied
- **shmdt** DOES NOT remove the shared memory segment from the system

Controlling shared memory

4) There also exists a control system call, that allows, among other things, to remove a shared memory region from the system, check its size, change its permissions ...

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- **shmid**: id returned by *shmget()*
- **cmd**: action to perform: `IPC_RMID`, `SHM_LOCK`, `SHM_UNLOCK`, `IPC_STAT`, `IPC_SET` ...
- **buf**: information

Controlling shared memory

- IPC_STAT can be used to get the status of the shared memory region
  - Specially useful to get the size when using a shared memory segment already created (in tha case we can pass 0 to *shmget* as the size)
- IPC_RMID Deletes the shared memory identifier from the system
  - It does not detach the shared memory from any of the processes using it
  - Should no process have this shared memory region attached, it will get removed
  - If any process has it attached it can continue to use it. The shared memory segment will get removed when it is no longer attached to any process
  - After this call (*shmctl(id,* `IPC_RMID` ...) any call to *shmget* with that key will consider that shared memory as non existant

Shared memory example

- The following function obtains a shared memory address given the key and the size (NULL in case of some error). If the option to create is specified the memory will be created unless it already exists, in which case in returns an error

```
void * ObtenerMemoria (key_t clave, off_t tam, int crear)
{
 int id;
 void * p;
 int flags=0666;

 if (crear)
   flags=flags | IPC_CREAT | IPC_EXCL;

 if ((id=shmget(clave, tam, flags))==-1)
   return (NULL);

 if ((p=shmat(id,NULL,0))==(void*) -1){
   if (crear)
      shmctl(id,IPC_RMID,NULL);
   return (NULL);
   }

 return (p);
}
```