List of Slides

1  Memory Management
4  Memory In Systems Design
5  Binding Times
6  Introduction to Memory Management
7  Raw Memory Model
8  Single User Contiguous Memory
9  Relocation - Why and How
10 Overlay Management
11 Protection
12 Fixed Partitions
13 Nonuniform Sized Fixed Partitions
14 Uniformly Sized Fixed Partitions
15 Simple Paging
16  Benefits of Simple Paging
17  Page Tables
18  Translation Lookaside Buffers
19  Hierarchical Address Caching
20  Dynamic Partitions
21  Fragmentation
22  Internal Fragmentation
23  External Fragmentation
24  Coalescing Holes
25  Compaction
26  Dynamic Partition Placement
27  Simple Segmentation
28  Memory Layout of A C Program
29  malloc
30  sbrk
31  Memory Heirarchy
32  Memory Heirarchy
33 Unix Process Control
34 Unix getid functions
35 Unix setid functions
36 Unix fork functions
38 The exec Family
40 exit, abort and atexit
Figure 1: Memory Connects CPU and Peripherals
Figure 2: Binding Times
4 – Introduction to Memory Management

Stallings [3] surveys the memory management issues:

1. Relocation
2. Protection
3. Sharing
4. Logical Organization
5. Physical Organization

Consider a series of solutions starting with the most primitive first [2].
5 – Raw Memory Model

The raw memory provides no services and gives the programmer complete control.

0

1 MB

Figure 3: Raw Machine Model
6 – Single User Contiguous Memory

Primitive operating systems (such as MS-DOS and CP/M) provide some interfaces to the hardware but not much else in the way of services.

Figure 4: Single User Contiguous Memory
Relocation refers to the ability to store a program at an arbitrary base memory address. Actual memory locations have physical or absolute addresses, user program’s access these locations using logical addresses.

Figure 5: Address Translation
Overlays have gone out of fashion with cheaper memory, users (and compilers) determine which code to swap in and out.

Figure 6: Overlay Management Example [2]
It is undesirable to permit user programs (accidentally or intentionally) to accesses memory outside of their partition.

![Diagram showing the division of memory into Monitor and User sections with a 32K shaded area representing resident monitor]

Figure 7: Protection in Resident Monitor Model
Fixed partitioning refers to memory being split into contiguous non-overlapping regions of precomputed sizes.

Fixed sized partitions make the selection of a partition for a job easy.
11 – Nonuniform Sized Fixed Partitions

Fixed partitions may have differing sizes.

Figure 8: Nonuniform Fixed Partitions [3]
12 – Uniformly Sized Fixed Partitions

Memory is frequently partitioned into uniformly sized regions.

![Diagram of uniformly sized fixed partitions]

Figure 9: Uniform Fixed Partition Allocation [3]
Paging provides relocation, and splits memory into fixed length partitions called *frames*.

Figure 10: Simple Paging [1]
14 – Benefits of Simple Paging

Simple paging allows discontiguous storage for memory objects exceeding the page frame size.

Figure 11: Simple Paging [1]
15 – Page Tables

One simple mechanism is to allocate some real memory space for a table, and hash page addresses using the high order address bits as pointers into the page table. There are 2 real memory accesses per virtual memory access.

![Diagram of page table mechanism]

Figure 12: Page Tables [1]
Translation lookaside buffers (TLB) eliminate one physical memory reference using special associative memory, which addressed by its contents in $O(1)$ parallel search time.

Figure 13: Associative Memory Lookup [1]
17 – Hierarchical Address Caching

Rather than placing all addresses in the TLB recently/frequently used addresses are stored in associative memory, with misses being serviced by the page table.

Figure 14: Hierarchical Address Caching [1]
18 – Dynamic Partitions

Dynamically partitioned memory allows placement of relocatable code in variable size contiguous memory regions.

Figure 15: Dynamic Partition Allocation [1]
Fragmentation makes available memory useless by breaking it into discontiguous pieces too small to use.

There are two categories of memory fragmentation:

1. *Internal Fragmentation* — A fixed partition contains more memory than required by the user, and some is wasted.

2. *External Fragmentation* — Results from the holes left by dynamic partitions.
Internal fragmentation occurs when fixed size partitions are too large.

Figure 16: Internal Fragmentation [3]
21 – External Fragmentation

External fragmentation happens when dynamic partitions are released. The fragments are frequently called holes.

Figure 17: External Fragmentation [1]
Adjacent holes in dynamic partitions should be coalesced into a single larger hole.

Figure 18: Coalescing Holes in Dynamic Partitions [1]
23 – Compaction

If the amount of memory available in the holes is large enough to service a request, the holes may made contiguous by compacting storage.

Figure 19: Compation of Dynamic Partitions [1]
24 – Dynamic Partition Placement

(a) FIRST-FIT STRATEGY
Place job in first storage hole on free storage list in which it will fit.

Free Storage List (Kept in storage address order, or sometimes in random order.)

<table>
<thead>
<tr>
<th>Start address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>14K</td>
</tr>
<tr>
<td>e</td>
<td>5K</td>
</tr>
<tr>
<td>g</td>
<td>30K</td>
</tr>
</tbody>
</table>

Request for 13K

(b) BEST-FIT STRATEGY
Place job in the smallest possible hole in which it will fit.

Free Storage List (Kept in ascending order by hole size.)

<table>
<thead>
<tr>
<th>Start address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>5K</td>
</tr>
<tr>
<td>c</td>
<td>14K</td>
</tr>
<tr>
<td>a</td>
<td>16K</td>
</tr>
<tr>
<td>g</td>
<td>30K</td>
</tr>
</tbody>
</table>

Request for 13K

(c) WORST-FIT STRATEGY
Place job in the largest possible hole in which it will fit.

Free Storage List (Kept in descending order by hole size.)

<table>
<thead>
<tr>
<th>Start address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>30K</td>
</tr>
<tr>
<td>a</td>
<td>16K</td>
</tr>
<tr>
<td>c</td>
<td>14K</td>
</tr>
<tr>
<td>e</td>
<td>5K</td>
</tr>
</tbody>
</table>

Request for 13K

Figure 20: Dynamic Partition Placements [1]
Segmentation provides relocation, and supports contiguous variable length partitions.
Segmentation often provides protection (counterexample Intel 8086).

Figure 21: Hardware Support for Simple Segmentation [2]
26 – Memory Layout of A C Program

Traditional Unix/C memory images of programs use segments.

Figure 22: Memory Layout of A C Program
Programmers often want to allocate data objects which persist beyond the function call creating them (e.g. constructors in OOP).

In C and C++ the `malloc` operator maintains a linked list of data objects, in the user program’s Data segment (on the Heap).

**Typical Malloc Implementation**

![Diagram of In Use List and Free List]

**Key**

- Memory management Information

**Figure 23: Typical Malloc Mechanism**
A user program can exhaust its default heap space allocation.

The Unix sbrk system call increases data segment allocation at run time.

Figure 24: Interaction between malloc and sbrk
Users want to:

1. Increase their address space, using slow cheaper memory to extend their more expensive faster memory.

2. Increase the speed at which they can the extended memory by using small amounts of expensive fast memory.
Figure 25: Hierarchical Memory
Unix supplies the following families of process control systems calls, including:

1. getid — (my own name) gets process identification information, e.g. getpid, getuid
2. setid — sets process identification information, e.g. setuid
3. fork — creates a runnable copy of the current process, e.g. fork, vfork
4. exec — replaces the current process image with another program, e.g. exec, excel, execve
5. exit — self termination, see also atexit()
6. wait — wait for a child process to terminate (see also wait4)
32 – Unix getid functions

A process can make the following identification inquiries:

1. int getpid(void) — the process’ id (pid).
2. int getppid(void) — the parent’s process id.
3. int getuid(void) — the real user id.
4. int geteuid(void) — the effective user id.
5. int getgid(void) — the real group id.
6. int getegid(void) — the effective group id.

Identification concerning file systems permissions can be either:

1. Real — the actual user/group of the account running the process or
2. Effective — temporary permissions modifications especially made for this user (e.g. setting your password).
3. Saved — saved by the exec function
33 – Unix setid functions

A process can change its id (and its security privileges, via):

1. `int setuid(int uid)` — sets the real user id (root/superuser only)
2. `int seteuid(int euid)` — sets the effective user id.
3. `int setgid(int gid)` — sets the real group id. (root/superuser only)
4. `int setegid(int egid)` — sets the effective group id.

These calls return the id passed or -1 if they fail due to either:

1. having an invalid id passed in.
2. attempting to change a real process id while not root or super user.
34 – Unix fork functions

The fork family creates new processes using the calls:

1. `int fork(void)` — returns the 0 if child or the pid of the newly created process.

2. `int vfork(void)` — child shares virtual memory and thread of control with the parent.

3. `int rfork(int flags)` — forks a process, using flags to regulate the resource copying used:
   (a) RPROC — if set, creates a new process, otherwise changes impact the current process.
   (b) RFNOWAIT — if set disassociates the child from the parent (so wait is not needed).
   (c) RFFDG — if set, copies the parents file descriptor table.
   (d) RCFDG — if set, creates a child with a
clean file descriptor table (mutually exclusive with RFFDG).

(e) RFMEM — if set, the kernel shares the entire memory space between the parent and child (data and bss segments). The stack segment is always split.

rfork allows more efficient process creation.
35 – The exec Family

Sometimes we don’t want the child to be an exact copy of the parent (e.g. starting the text editor from the shell). To do this we use the exec family of functions:

1. int execvp()
2. int execvp()
3. int execl()
4. int execv()
5. int execle()
6. int execve()

The suffix letters denote the following:

1. l — passes each command line argument with an individual pointer
2. v — passes the command line arguments using a vector of characters
3. p — searches directories in the order of the PATH environment variable.
4. e — uses an environment vector to define environment variables.

Figure 8.6 on page 411 of Stevens is particularly helpful.
Exiting a U*ix process can be done in a variety of ways.

1. `abort(void) — system call coerces a core dump of the program.

2. `exit(int status) — terminates a program, returning status to the operating system, invoking functions registered using `atexit` in reverse order of registration. Additionally `exit` —

   (a) Closes all open file descriptors

   (b) Notifies (via the HUP signal) the children of the exiting process.

   (c) If the parent is not waiting, turns the calling process into a zombie.

   (d) Memory mapped objects are detached.

   (e) System V IPCS are detached (shm, semaphores, etc).

   (f) Locks on data are released.
3. int atexit(void (*func)(void)) —
    Registers functions (in LIFO order) to be invoked when exit is invoked.

Many unix implementations use a supporting function _exit, which normally should not be called by application programs.
References

