COS222

Lecture 31:

Overview of the MINIX 3 Process Manager.

Part 1 of 2

Overview of the MINIX 3 Process Manager.

- Memory management in MINIX 3 is simple:
  - paging is not used at all.
  - swapping is not used at all.

- Swapping code is available in the complete source and could be activated to make MINIX 3 work on a system with limited physical memory.

Overview of the MINIX 3 Process Manager.

- We will study a user-space server designated the **process manager** (**PM**).
- The process manager handles system calls relating to process management.
- Of these some are intimately involved with memory management.
  - The fork,
  - exec,
  - and brk calls
  - are in this category.
- Process management also includes
  - processing system calls related to signals,
  - and also handles setting and querying the real time clock

Overview of the MINIX 3 Process Manager.

- In MINIX 3 the process management and memory management are merged into one process
  - It is possible that in a future release of MINIX, process management and memory management will be completely separated
    - It really should be
- The PM maintains a list of holes sorted in numerical memory address order.
  - When memory is needed, either due to a fork or an exec system call, the hole list is searched using first fit for a hole that is big enough.
  - Without swapping, a process that has been placed in memory remains in exactly the same place during its entire execution.
  - It is never moved to another place in memory, nor does its allocated memory area ever grow or shrink.

Overview of the MINIX 3 Process Manager.

- This strategy for managing memory is somewhat unusual and deserves some explanation:
  - **1.** The desire to keep the system easy to understand.
  - **2.** The architecture of the original IBM PC CPU (an Intel 8088),
    - As it had no MMU, so including paging was impossible to start with.
  - **3.** The goal of making MINIX 3 easy to port to other hardware,
- MINIX 3 is targeted to some extent at low-end systems such as embedded systems.
  - Nowadays, digital cameras, DVD players, stereos, cell phones, and other products have operating systems, but certainly do not support swapping or paging.
  - MINIX 3 is quite a reasonable choice in this world.
    - This was true when the book was written, this doesn't hold true for all smart phones and/tablets nowadays.

Overview of the MINIX 3 Process Manager.

- Another way in which implementation of memory management in MINIX 3 differs from that of many other operating systems.
  - The PM (and MM) is not part of the kernel.
  - Instead, it is a process that runs in user space and communicates with the kernel by the standard message mechanism.
- This an example of the separation of **policy** and **mechanism.**
  - **Policy**: The decisions about which process will be placed where in memory are made by the PM.
  - **Mechanism:** The actual setting of memory maps for processes is done by the system task within the kernel
- This split makes it relatively easy to change the memory management policy without having to modify the lowest layers of the operating system

Memory Layout

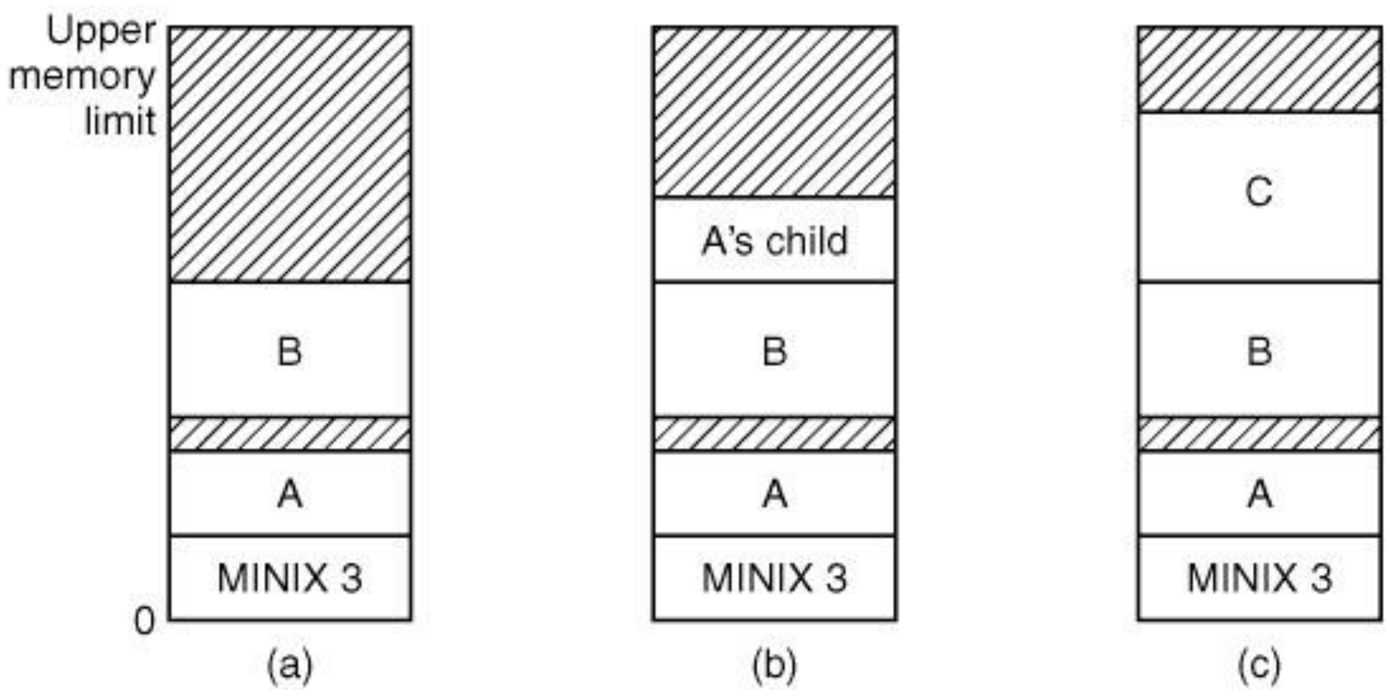
- MINIX 3 programs may be compiled to use **combined I and D space**, (instruction and data)
  - in which all parts of the process (text, data, and stack) share a block of memory which is allocated and released as one block.
- This was the default for the original version of MINIX. In MINIX 3, however, the default is to compile programs to use **separate I and D space.**
- The later being more complicated.

Memory Layout

- We will discussed the simpler combined approach first.
- In normal MINIX 3 operation memory is allocated on “two” occasions:
  - First, when a process forks,
    - the amount of memory needed by the child is allocated
  - Second, when a process changes its memory image via the exec system call,
    - the space occupied by the old image is returned to the free list as a hole, and memory is allocated for the new image. (not necessarily using the deallocated space)
  - There is a third case (though rare): a system process can request memory for its own use;
    - for instance, the memory driver can request memory for the RAM disk. This can only happen during system initialization.
- Memory is also released whenever a process terminates, either by exiting or by being killed by a signal

Memory Layout

- Figure below shows memory allocation during a fork and an exec.



- (a) Originally. (b) After a fork. (c) After the child does an exec.

Memory Layout

- Doing this kind of memory management is not trivial.
  - Consider the possible error condition that there is not enough memory to perform an exec.
  - A test for sufficient memory to complete the operation should be performed before the child's memory is released, so the child can respond to the error somehow.
  - This means the child's memory must be considered as if it were a hole while it is still in use.

Memory Layout

- When memory is allocated, either by the fork or exec system calls, a certain amount of it is taken for the new process.
  - In the former case, the amount taken is identical to what the parent process has.
  - In the latter case, the PM takes the amount specified in the header of the file executed.
  - Once this allocation has been made, under no conditions is the process ever allocated any more total memory.

Memory Layout

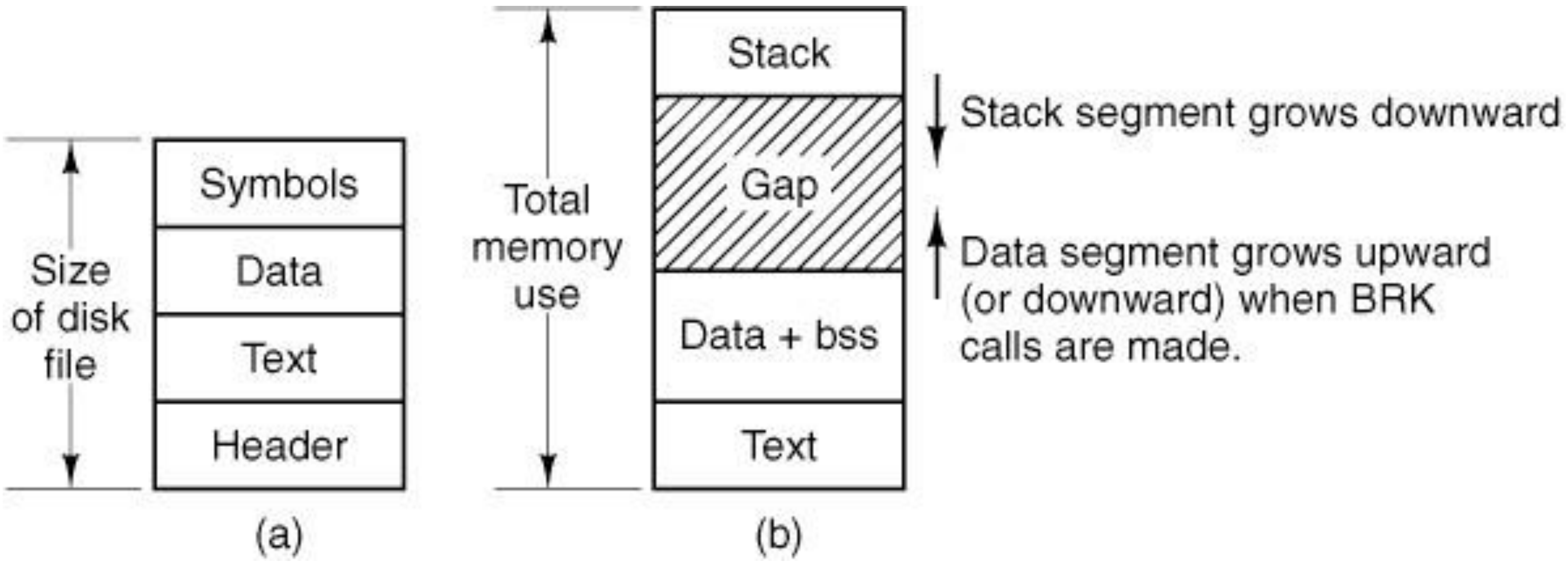
- **Separate I and D space:**
- Programs with separate I and D space take advantage of an enhanced mode of memory management called **shared text.**
  - When such a process does a *fork*, only the amount of memory needed for a copy of the new process' data and stack is allocated.
  - Both the parent and the child share the executable code already in use by the parent

Memory Layout

- When such a process does an exec,
  - the process table is searched to see if another process is already using the executable code needed.
  - If one is found, new memory is allocated only for the data and stack, and the text already in memory is shared.
- Shared text complicates termination of a process.
  - When a process terminates it always releases the memory occupied by its data and stack.
  - But it only releases the memory occupied by its text segment after a search of the process table reveals that no other current process is sharing that memory

Memory Layout

- Loading a program from disk file to internal memory:


Size of disk file
Symbols
Data
Text
Header
(a)
Total memory use
Stack
Gap
Data + bss
Text
Stack segment grows downward
Data segment grows upward (or downward) when BRK calls are made.
(b)

Memory Layout

- The header on the disk file contains information about the sizes of the different parts of the image, as well as the total size.
  - In the header of a program with common I and D space, a field specifies the total size of the text and data parts;
    - these parts are copied directly to the memory image.
  - The data part in the image is enlarged by the amount specified in the *bss* (historically Block Started by Symbol) field in the header.
    - This area is cleared to contain all zeroes and is used for uninitialized static data.
  - The total amount of memory to be allocated is specified by the *total* field in the header.
  - The symbol table is used for debugging and is not copied into memory.

Memory Layout

- If, for example, a program has 4 KB of text, 2 KB of data plus bss, and 1KB of stack, and the header says to allocate 40 KB total, the gap of unused memory between the data segment and the stack segment will be 33 KB.
- It is possible to change the amount of memory available for expansion on initialization
  - E.g. *chmem =10240 a.out*
- which changes the header field so that upon exec the PM allocates a space 10240 bytes more than the sum of the initial text and data segments.

Memory Layout

- For a program using separate I and D space (indicated by a bit in the header that is set by the linker), the total field in the header applies to the combined data and stack space only.
  - So not including the text segment.
- A program with 4 KB of text, 2 KB of data, 1 KB of stack, and a total size of 64 KB
  - will be allocated 68 KB
    - (4 KB instruction space (text), 64 KB stack and data space),
    - Meaning there is 64-2-1=61K space for the stack and data segments to grow.

Message Handling

- Like all the other components of MINIX 3, the process manager is message driven.
- After the system has been initialized, PM enters its main loop,
  - which consists of waiting for a message,
  - carrying out the request contained in the message,
  - and sending a reply.

Message Handling

- Two message categories may be received by the process manager.
  - For high priority communication between the kernel and system servers such as PM,
    - a **system notification message** is used
    - The details of which are discussed in 4.8.
  - The majority of messages received by the process manager result from system calls originated by user processes.

Message Handling

- For this category, the next figure gives the list of
  - legal message types,
  - input parameters,
  - and values sent back in the reply message.

Message Handling

Message type	Input parameters	Reply value
fork	(none)	Child's PID, (to child: 0)
exit	Exit status	(No reply if successful)
wait	(none)	Status
waitpid	Process identifier and flags	Status
brk	New size	New size
exec	Pointer to initial stack	(No reply if successful)
kill	Process identifier and signal	Status
alarm	Number of seconds to wait	Residual time
pause	(none)	(No reply if successful)
sigaction	Signal number, action, old action	Status
sigsuspend	Signal mask	(No reply if successful)
sigpending	(none)	Status
sigprocmask	How, set, old set	Status
sigreturn	Context	Status
getuid	(none)	Uid, effective uid
getgid	(none)	Gid, effective gid
getpid	(none)	PID, parent PID

Message Handling

Message type	Input parameters	Reply value
setuid	New uid	Status
setgid	New gid	Status
setsid	New sid	Process group
getpgrp	New gid	Process group
time	Pointer to place where current time goes	Status
stime	Pointer to current time	Status
times	Pointer to buffer for process and child times	Uptime since boot
ptrace	Request, PID, address, data	Status
reboot	How (halt, reboot, or panic)	(No reply if successful)
svrctl	Request, data (depends upon function)	Status
getsysinfo	Request, data (depends upon function)	Status
getprocnr	(none)	Proc number
memalloc	Size, pointer to address	Status
memfree	Size, address	Status
getpriority	Pid, type, value	Priority (nice value)
setpriority	Pid, type, value	Priority (nice value)
gettimeofday	(none)	Time, uptime

Process Manager Data Structures and Algorithms

- Two key data structures are used by the process manager:
  - the process table.
  - and the hole table.
- We will begin with a discussion of the former.
- Some process table fields are
  - needed by the kernel,
  - others by the process manager,
  - and yet others by the file system.
- In MINIX 3, each of these three pieces of the operating system has its own process table, containing just those fields that it needs.

Process Manager Data Structures and Algorithms

- With a few exceptions, entries correspond exactly, to keep things simple.
  - Thus, slot *k* of the PM's table refers to the same process as slot *k* of the file system's table.
  - When a process is created or destroyed, all three parts update their tables to reflect the new situation, in order to keep them synchronized.

Process Manager Data Structures and Algorithms

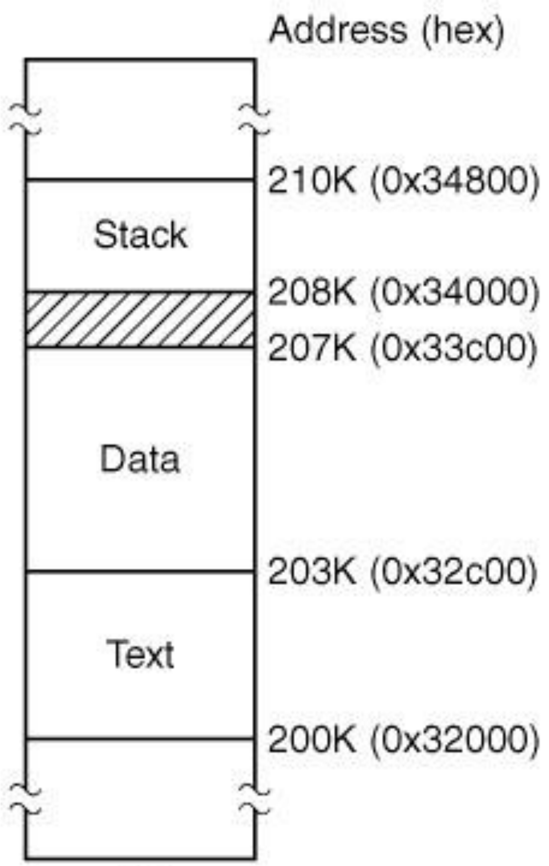
- The exceptions are processes that are not known outside of the kernel,
  - either because they are compiled into the kernel, like the *CLOCK* and *SYSTEM* tasks,
  - or because they are place holders like *IDLE*, and *KERNEL*.
- In the kernel process table their slots are designated by negative
- numbers. These slots do not exist in the process manager or file system process tables.

Processes in Memory

- The PM's process table is called *mproc* and its definition is given in *src/servers/pm/mproc.h*.
- It contains all the fields related to a process' memory allocation, as well as some additional items.
    - The most important field is the array *mp_seg*, which has three entries,
        - for the text,
        - data,
        - and stack segments, respectively.
- Each entry is a structure containing the
    - virtual address,
    - physical address,
    - and length of the segment,
- all measured in **clicks** rather than in bytes.

Processes in Memory

- The size of a click is implementation dependent.
  - In early MINIX versions it was 256 bytes.
  - For MINIX 3 it is 1024 bytes.
  - All segments must start on a click boundary and occupy an integral number of clicks.



(a)

	Virtual	Physical	Length
Stack	0x8	0xd0	0x2
Data	0	0xc8	0x7
Text	0	0xc8	0

(b)

	Virtual	Physical	Length
Stack	0x5	0xd0	0x2
Data	0	0xcb	0x4
Text	0	0xc8	0x3

(c)

- Figure (a) A process in memory.
- (b) Its memory representation for combined I and D space.
  - In this model, the text segment is always empty, and the data segment contains both text and data.
- (c) Its memory representation for separate I and D space

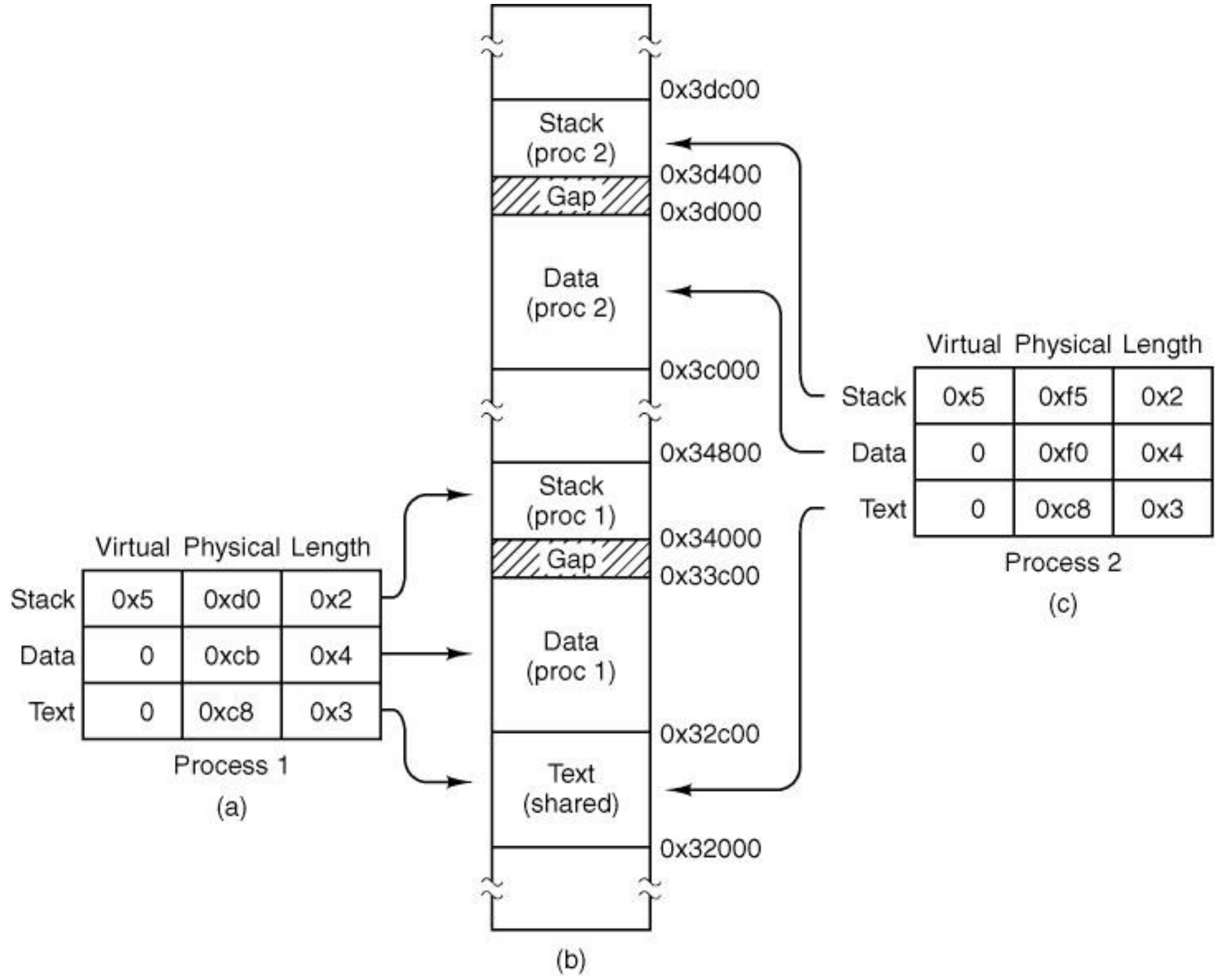
Shared Text

- The contents of the data and stack areas belonging to a process may change as the process executes,
  - but the text does not change.
- It is common for several processes to be executing copies of the same program,
  - for instance several users may be executing the same shell.
- Memory efficiency is improved by using **shared text**.

Shared Text

- When exec is about to load a process, it opens the file holding the disk image of the program to be loaded and reads the file header.
  - If the process uses separate I and D space we can utilize shared text.
- If a process in memory is found to be executing the same program that is about to be loaded, there is no need to allocate memory for another copy of the text.
  - So only the data and stack portions are set up in a new memory allocation.

Shared Text



	Virtual	Physical	Length
Stack	0x5	0xd0	0x2
Data	0	0xcb	0x4
Text	0	0xc8	0x3

Process 1

(a)

	Virtual	Physical	Length
Stack	0x5	0xf5	0x2
Data	0	0xf0	0x4
Text	0	0xc8	0x3

Process 2

(c)

(b)

The Hole List

- The other major process manager data structure is the **hole list** (book states hole table, which is a mistake):
  - which lists every hole in memory in order of increasing memory address.
  - The gaps between the data and stack segments are not considered holes; they have already been allocated to processes.
    - Consequently, they are not contained in the free hole list.

The Hole List

- Each hole list entry has three fields:
  - the base address of the hole, in clicks;
  - the length of the hole, in clicks; and
  - a pointer to the next entry on the list.
- The list is singly linked, so it is easy to find the next hole starting from any given hole, but to find the previous hole, you have to search the entire list from the beginning until you come to the given hole

The Hole List

- The reason for recording everything about segments and holes in clicks rather than bytes is simple: it is much more efficient in terms of space.
  - Greater range
  - However lower precision.
- To allocate memory,
  - the hole list is searched, starting at the hole with the lowest address, until a hole that is large enough is found (first fit).
  - The segment is then allocated by reducing the hole by the amount needed for the segment, or in the rare case of an exact fit, removing the hole from the list.

The Hole List

- This scheme is fast and simple but suffers from both a small amount of internal fragmentation (up to 1023 bytes may be wasted in the final click, since an integral number of clicks is always taken) and external fragmentation.