# Lecture 8

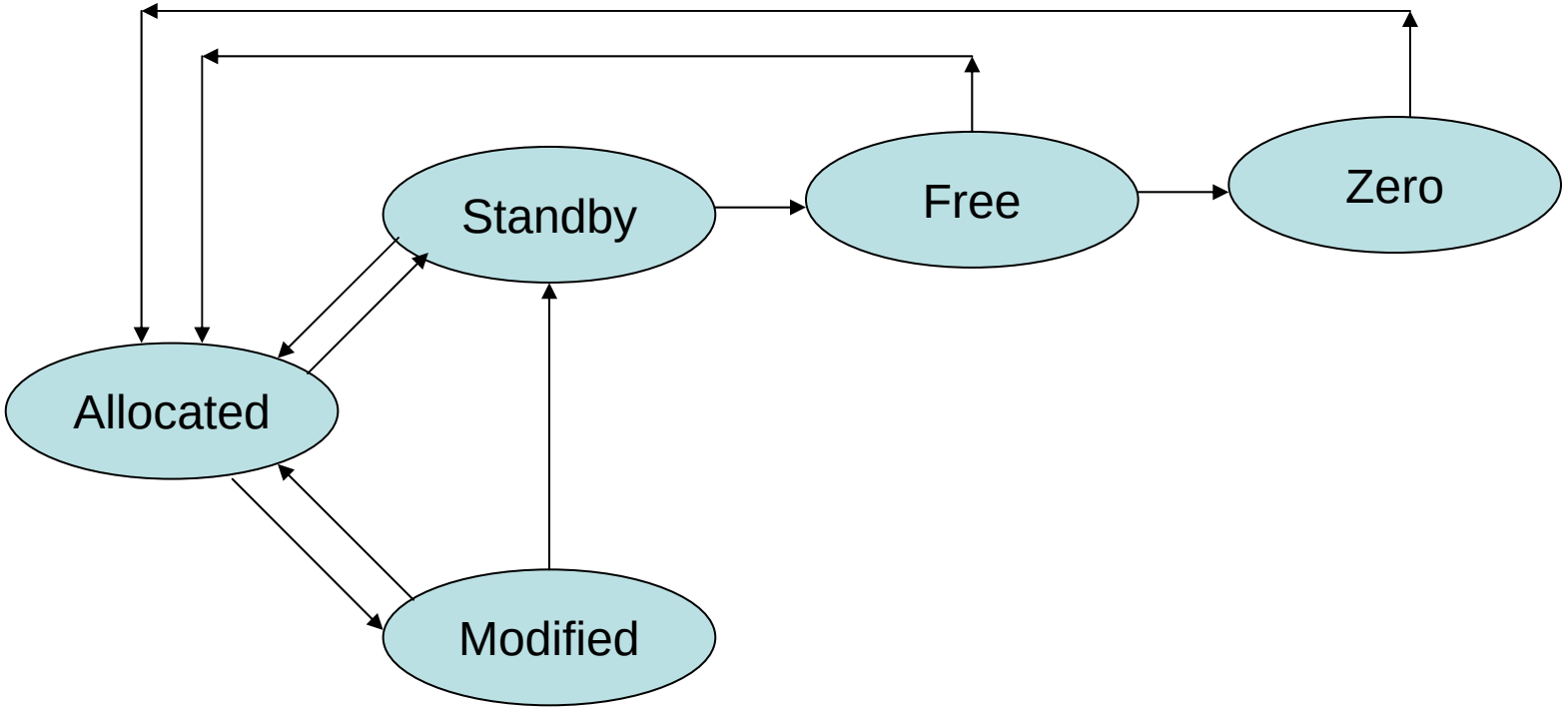
## Examples of memory management

# Windows NT

- Prior to NT, Win32 systems did not limit the access of processes to only the memory allocated to them. As a result, processes often had dangerous access to other processes and the OS.
- At the lowest level, Win32 memory calls operate on pages:
  - *VirtualAlloc* and *VirtualAllocEx* are called for allocation;
  - *VirtualFree* and *VirtualFreeEx* calls release memory;
  - *VirtualLock* forbids pages to be swapped;
  - *CreateFileMapping* is used both to map files to memory and to establish memory areas shared by multiple processes.

# Page management

- Win NT uses a working set page management strategy – each process has an admin determined *min and max working set size*. The default values are based on the amount of physical memory in the system.
- When a process needs to read a page and is using its max working set size, then the choice of page to be replaced comes from its set. If it needs to read a page and its set size is below the min, one page will be added to it.
- To keep available free pages, the system periodically checks the number of free pages and if there are too few, it runs the working set manager. This manager trims the working sets that are above their min size and frees pages.
- Regarding the page replacement policy, the system initially used FIFO: the page selected for removal is the one that has been in the working set the longest.
- However, removed pages are placed on a list from where they can be re-added to the working set if they are referenced before their frames are re-allocated.
- Recently, the policy in use is NFU: pages are periodically scanned and if they were accessed since the last scan, a counter is incremented. The counter is also aged so as to weigh recent access more heavily than older accesses. Then, pages are added to a list from which they can be reallocated if needed.




Windows NT page frame state machine

# TinyOS

- Embedded systems have small main memories and no memory management hardware.
- There are no memory allocation features of the OS.
- The layout of components in memory is determined at build-time by the linker.

# Linux memory management

- To support processes that may allocate memory implicitly as part of the process creation and as a result of stack growth, or explicitly through two system calls, Linux implements a number of mechanisms that operate both on *fixed-sized pages* and *variable-sized blocks*.
- Variable allocation can be in the form of multiple contiguous pages or in the form of smaller allocations within pages.
- Linux supports *demand paging* and *page swapping*.

# Linux physical memory layout

- The physical memory layout for any system is highly dependent of the hardware as well as the design of the OS.
- For Intel x86, the 1st MB is largely unused except during booting and some 360 KB used for accessing memory-mapped I/O controllers. The next several MB store the uncompressed kernel image.
- The area between the top of the kernel and the end of the first 16 MB area is used primarily for I/O buffers ZONE_DMA.
- The remainder of the memory  is called ZONE_NORMAL.
- For systems with more than 1GB of physical memory, ZONE_NORMAL ends at 896MB, and the rest is ZONE_HIGHMEM.

# System calls

- Process management calls have implications for memory management: *fork()* copies the parent space; this is done using copy-on-write (COW) which permits sharing of code segments.
- *execve()* requires the release of the memory previously used by the process, followed by memory allocation to the new process. A feature of Linux is that it does not load the executable code into memory as a result of *execve().* Rather, it treats the executable as a memory-mapped file, and then uses demand paging to load pages as needed.
- The UNIX *brk()* system call has a parameter that specifies the first address that is not part of the data segment. The interpretation is that everything else is available for the stack.
- In Linux, however, the breakpoint defines the separation between data and the area used by *mmap().*
- Finally *mmap()* is for mapping files in memory.

# Allocation mechanisms

- Linux uses the Buddy algorithm to effectively allocate and deallocate blocks of pages. The page allocation code attempts to allocate a block of one or more physical pages.
- The allocation algorithm first searches for blocks of pages of the size requested. It follows the chain of free pages that is queued on the *list* element of the free_area data structure. If no blocks of pages of the requested size are free, blocks of the next size (which is twice that of the size requested) are looked for. This process continues until all of the free_area has been searched or until a block of pages has been found.
- If the block of pages found is larger than that requested it must be broken down until there is a block of the right size. The free blocks are queued on the appropriate queue and the allocated block of pages is returned to the caller.
- Allocating blocks of pages tends to fragment memory with larger blocks of free pages being broken down into smaller ones. The page deallocation code recombines pages into larger blocks of free pages whenever it can. In fact the page block size is important as it allows for easy combination of blocks into larger blocks.
- Whenever a block of pages is freed, the adjacent or buddy block of the same size is checked to see if it is free. If it is, then it is combined with the newly freed block of pages to form a new free block of pages for the next size block of pages.

# Slab allocator

- Not all allocation requests are multiple of pages, or even a full page. Sometimes, a data structure of few tens of bytes needs memory space.
- Linux implements a mechanism called *the slab allocator*. Slabs are collections of free memory blocks of a particular size. When a request matches that size, the slab can satisfy it. If the slab is empty, one or more pages are divided into blocks of the required size and added to the slab.
- When a block is released, it is added to the slab.

# Page management

- For portability, Linux abstracts its page management – it uses a four-level page table design.
- The uppermost bits of the VA index a page global directory (pgd). The selected entry points to a page upper directory (PUD), which is indexed by the next most significant bits of the VA.
- The pud entry points to a page mid-level directory (pmd) indexed by the third group of bits.
- Finally, the pmd entry points to a page table (pt) indexed by the fourth group of bits. The least significant group of bits of the VA give the offset into the page frame pointed to by the selected page table entry, which includes the page frame number part of the physical address (PA).
- The basic design does not specify the number of bits used for each of the fields in either VA or the PA.
- For hardware (i.e. Intel x86) that has fewer levels of page tables, Linux merges levels together – i.e. pud and pmd are merged into the pgd. Both pgd and pt are indexed by 10 bits of the 32-bit VA. This leaves 12 bits to give the page offset, resulting a page size of 4096 bytes.

# Memory Review

- **Free space management**
  - Free bitmap;
  - Linked list;
  - Binary tree;
  - Hash table.
- **Memory allocation**
  - First fit;
  - Next fit;
  - Best fit;
  - Worst fit;
  - Buddy system;
  - Swapping;
- **Page replacement**
  - FIFO;
  - Second chance;
  - Not recently used (NRU);
  - Least recently used (LRU);
  - Not frequently used (NFU);
  - The working set.

# Analysis

- Different mechanisms have different cost in terms of resources that are used and execution time.
- What is the main parameter of interest for memory allocation strategies ?
- Consider your own design of a memory management system (MMS) for a multi-process operating system. Would you choose the same mechanisms as Linux ? Explain your options.
- Regarding its implementation, will the MMS be a multi-process system itself ?