[redacted]

Virtual Memory Management

[redacted]

[redacted]

Page table Implementation

- The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**, if the page table is reasonably small (e.g., 256 entries). However usually, contemporary computers allow the page table to very large (e.g., 1 million entries), **so the page table is kept in main memory.** It is also important to note that a page table can itself be paged out. It is possible that when we go to use a portion of a page table it may not be in main memory. We would then need to swap it in just like any other portion of memory.
- **Page-table base register (PTBR)** is used to point to the page table. So changing page tables requires changing only this one register. That substantially reduces context-switching time.
- The problem with this approach is that it requires **2 memory accesses**: 1 for the page table entry, 1 for the data/instruction. **Thus, memory access is slowed by a factor of 2!**
- The solution to above problem is to use a special, small, fast-lookup hardware cache, called a **translation look-aside buffer (TLB).** The TLB is **associative**, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. It searches by key field and if the key is found it returns the corresponding value field. Very **fast**, hence is **expensive**.
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and **is used to provide address-space protection for that process**. When the TLB attempts to resolve virtual page numbers, it makes sure that the ASID for the currently running process matches the ASID associated with the virtual page. If ASIDs do not match, it treated as a TLB miss. In addition to providing address-space protection, **an ASID allows the TLB to contain entries for several different processes simultaneously.** If the TLB does not support for separate ASIDs, every time a new page table is selected (each context switch), the TLB must be **flushed** to ensure that the next executing process does not use the wrong translation information.

TLB is Associative Memory

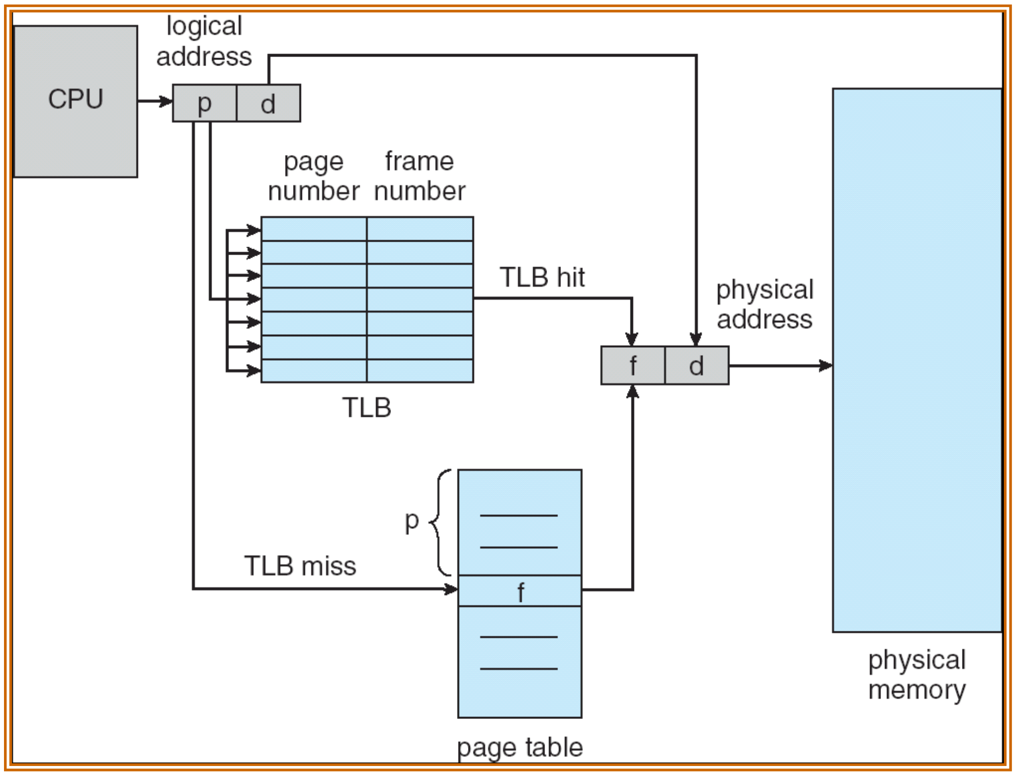
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

Hit ratio – percentage of times that a page number is found in the TLB. E.g. 80% hit ration means that we find the desired page number in the TLB 80% of the time.

For example:

- Associative Lookup(time takes to search the TLB) = T time unit
- Assume memory cycle time (to access memory) is 1 microsecond
- Let's denote Hit ratio = H
- Then to find **Effective Memory- Access Time** (EAT):

EAT = (1(memory access) + T) * H + (2 * 1(memory access) + T) * (1 – H)

$\underline{H}+\underline{T*H} + 2 - \underline{2*H} + T - \underline{T*H} = \mathbf{2 + T - H}$

__**How does malloc work:**__

What we know:

- We already know that each process has a heap. This heap spans the virtual address space
- The boundaries of the heap are denoted by “end” and “brk” in c
- Within this span of virtual memory, all of our malloced data is stored

What is malloc:

- malloc is part of a **user-level library**
- This library keeps **data structures** that maintain a **list** of all available chunks of memory currently in the heap
- Calling malloc causes an interaction with these data structures. malloc will search for a chunk of memory of adequate size. If a match is found malloc updates the data structures to reflect that the chunk of memory is no longer available and also **records** how much **actual space** was taken up. Finally, malloc returns a **pointer** to this newly allocated memory so that you may now use it.
- When malloc cannot find a big enough chunk of available memory in its data structures it must get more memory. Malloc is able to grow the heap using the **brk()** system call. Malloc has to use this system call because there is no way to ask for additional pages without seeking the OS help. Only the OS memory manager can allocate new physical pages to a process.

Freeing memory:

- Dynamically allocated memory can be freed using the **free()** function
- When free is given a **pointer** to a chunk of memory it is able to retrieve how big the chunk is and then can add that memory back to the list of available memory chunks

Brk System Call:

- There are two ways to use the break system call. The first way is to manually call brk() itself. The second way is to call sbrk(), which is a user-level function. This function then internally calls brk().

int brk(void *addr)

- This system call sets the program break of the process (the end of the data) to addr. This system call succeeds as long if **addr** is a reasonable value. The system also needs to have enough memory as well as the process.

void *sbrk(intptr_t increment)

- This system call increments the **program break** of the process by **increment** bytes. It returns a **pointer** to the start of the newly allocated memory. Passing in an increment value of 0 has the same effect as requesting a pointer to the **current program break**.

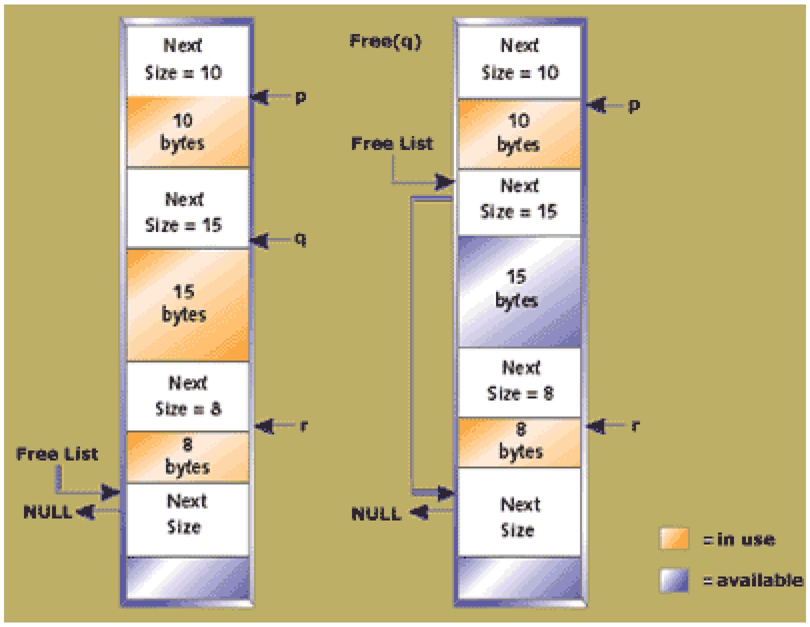


Figure 1 - Example of what malloc's data structures may represent

Dynamic Memory Storage Allocation Strategies:

-Given the situation of having a list of available memory chunks and wanting to allocate a space of size n, we have a couple of different strategies

First Fit:

Allocates the first available chunk that is big enough (> n)

Best Fit:

- Allocates the **smallest chunk of memory** that is large enough to hold our desired space requirement.
- One downside of this method is that the **whole list** of available memory must be searched in order to find the smallest "good" chunk. This downside can be avoided if the list is ordered by size
- This method also produces the **smallest chunk leftover** from splitting the chosen chunk into the memory needed and the memory that is leftover.

Worst Fit:

- Allocates the **largest chunk** of memory.
- This method has the same downside of needing to search the **entire list** if it is not ordered by size
- Opposite to Best Fit, this method produces the **largest chunk lefto**ver from splitting the chosen memory chunk into memory needed and leftover memory.

*In terms of storage utilization and speed, worst fit is not as good of a choice as best fit and first fit.