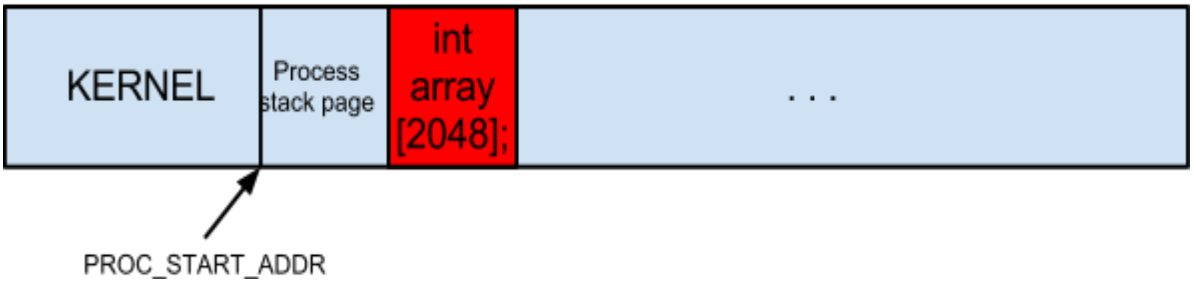
Section Notes: Week of 10/29

- Virtual Memory Beyond Asst 3: Page Faults and Paging/Swapping
- TLB/Caching/Page Lookups
- Utilization

Memory beyond Asst3

Page Faults: There are a few different types of page faults. One of them your WeensyOS from Asst 3 can handle. This is the page fault that occurs when the user tries to write to a virtual page that **has not been allocated** yet.
For example, in Asst 3, only the stack page has been allocated for each process initially. But maybe the processes want to initialize and use an enormous array that is larger than the stack page and overflows into the next page:



The next page is red because it is unallocated, so when the process starts trying to write to this array, a PAGE FAULT is incurred, because that page is unallocated!!! A context switch into the kernel happens (trap), to handle this page fault. To fix the fault, the kernel simply allocates another page (for example using sys_page_alloc as was written in Asst 3), and maps that physical page to the virtual page of the array. Page fault fixed! The kernel passes control back to the user. That was an easy fault to handle.

Another type of page fault is more complicated and is related to the paging system. There is no paging system in Asst 3, but in real operating systems, paging is an essential part of the virtual memory layer. In real OSes, the physical memory system is much larger than actual RAM, which is the resident memory array on the actual motherboard. Therefore, the OS gives the illusion/provides the abstraction (same thing) that all physical pages are actually resident in memory. But in fact, because there are more physical pages than can fit in RAM, some of the physical pages are actually sitting on DISK at any one point. This area on disk is called the **swap space**. The process of bringing pages into RAM from the swap space is called **paging**. Physical pages can only be used if they are resident in RAM, which means that if they are sitting in the swap space, they must be paged in, which leads to...

The second type of page fault, which occurs when the physical page required is sitting in swap space. Then to handle this fault, the kernel has to bring the page in from swap space. There are more complications involved in this operation such as-- if the RAM is full-- which page needs to be swapped out? This leads to the question of how we store the page lookups, that tell us which

pages are resident in RAM. Which leads us to:

TLB/Caching/Page table lookups

We are given a system with the following properties:

- The memory is byte addressable.
- Memory accesses are to 4-byte words
- Physical addresses are 16 bits wide.
- Virtual addresses are 20 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

In the following tables, **all numbers are given in hexadecimal.** The contents of the TLB and the page table for the first 32 pages are as follows:

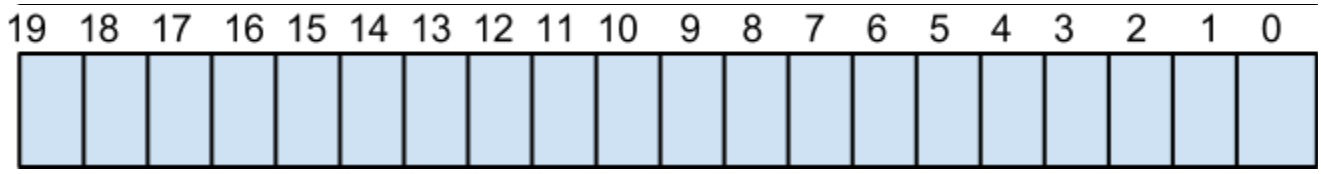
TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

Question 1a: The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

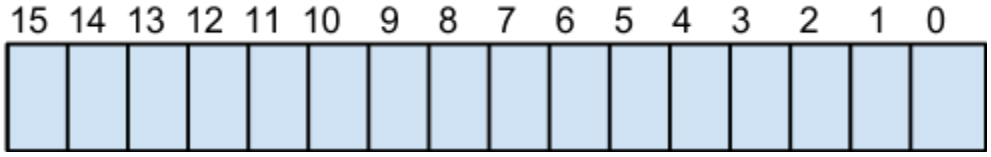
VPO The virtual page offset

VPN	The virtual page number
TLBI	The TLB index
TLBT	The TLB tag



Question 1b: The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

PPO	The physical page offset
PPN	The physical page number

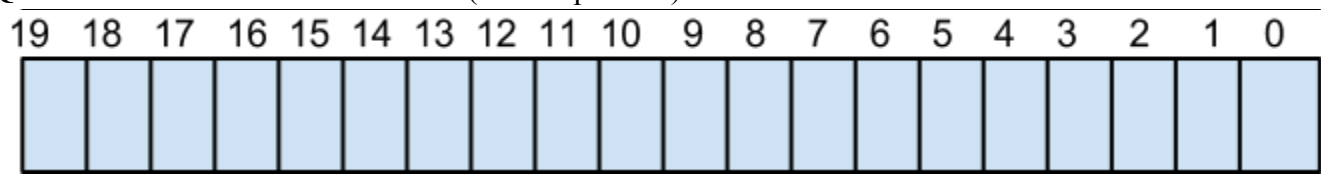


For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: 0x`7E37C`

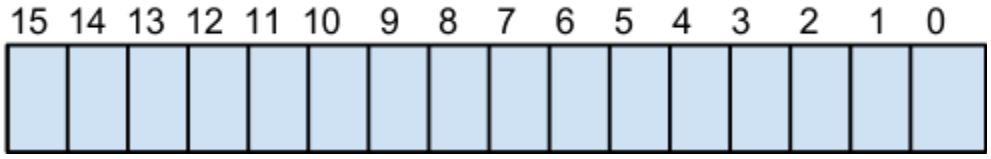
Question 2a: Virtual address format (one bit per box)



Question 2b: Address translation

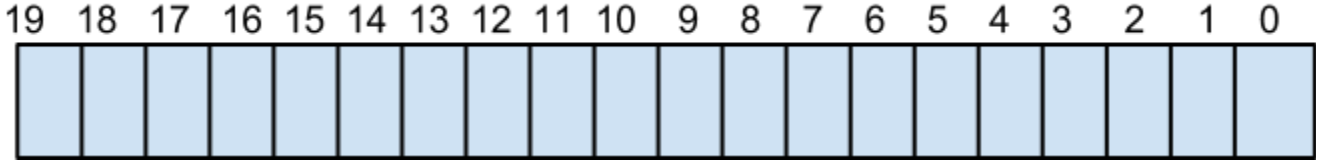
Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

Question 2c: Physical address format (one bit per box)



Virtual address: `0x16A48`

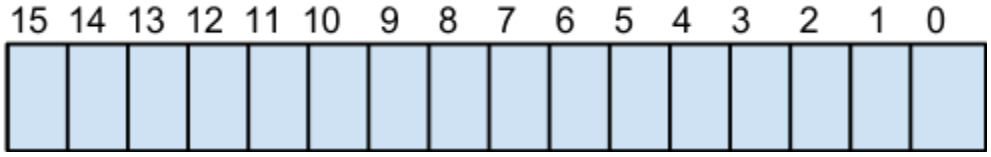
Question 3a: Virtual address format (one bit per box)



Question 3b: Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

Question 3c: Physical address format (one bit per box)

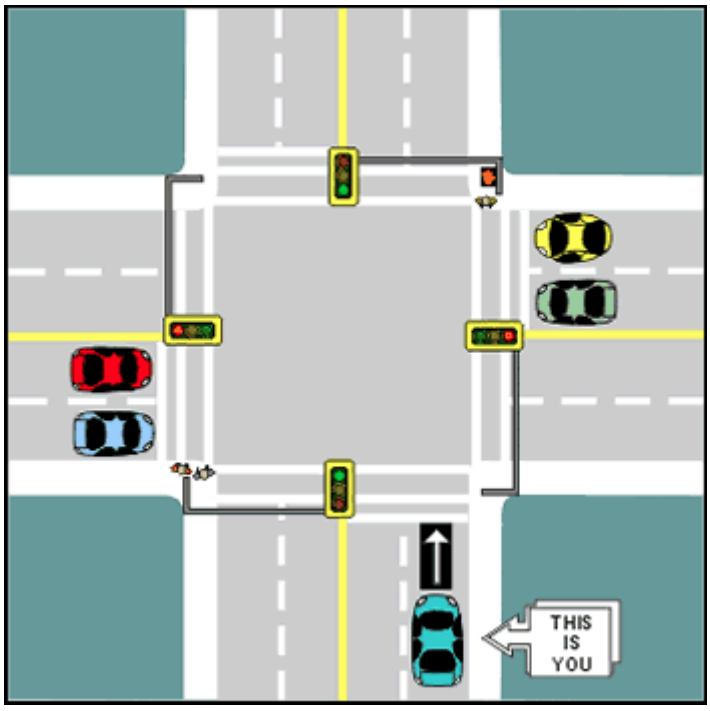


Utilization

Utilization is a study of how well a system's resources are used. Because this class is mostly confined to talking about a single machine, in our context, we mean utilization in the sense of the resources of a single computer. And the two largest resources of a computer are its CPU and memory. (In a distributed system, you might also consider the network as a resource.)

CPU as a resource

Example: Imagine a 4-way traffic light intersection, where "useful work" is defined by cars actually crossing the intersection (indeed, this achieves "progress" in the normal sense as well as in our contrived example sense). Then each street going into the intersection can be thought of as a queue of "tasks", where each car is a task.



Then the parameter here that will affect the utilization of the intersection is the time in between traffic light changes. Namely, imagine that the time in between red lights is 1 sec. Then this ensures that all lanes get to go (and let's just assume that one car can cross the intersection in 1 second), but it means that every lane advances very slowly. Or, imagine that the time in between red lights is 10 seconds. Then every time a lane gets to go, the cars go very smoothly and you can advance many vehicles before needing to switch to the next lane.

At this point, you can probably already see the analogy we are drawing with the CPU. The 4 lanes are 4 **processes** that require CPU time, the **CPU** is the intersection, and the lights signal for **context switches** in between the processes. The problem of the intersection is a problem of CPU utilization and process scheduling.

More complex: Now go back to the case where the time between red lights is hard-coded to be 10 seconds. What happens if a lane has no cars, but one of the lanes has a huge line? For example, let's say everyone is evacuating Manhattan for Hurricane Sandy, and everyone is going West into Jersey. It doesn't make much sense to let each lane get the same amount of time, and in a computer this is also acknowledged. Therefore, you can have "sleeping" processes, that don't get any time on the CPU; sleeping processes are not eligible to be scheduled. You can take a look at the different states a process might have in the procstate_t struct from Asst 3.

NB: Note that these process states are also a mechanism to increase CPU utilization; otherwise, you could never mark states as "blocked" or "sleeping", in which case you would have to schedule all processes on the CPU, regardless of their need for the CPU. Then a process that is waiting for a large disk I/O will waste CPU time, because when it is scheduled

on the CPU, it doesn't do any useful work. This is analogous to the traffic intersection case where all lanes get 10 seconds in the intersection, but only 1 lane has any cars (I'm sure you've encountered this frustration many times while driving. It is silly.).

How Device I/O affects CPU utilization

Any time there is some sort of latency in the system, you should always be thinking about CPU utilization. Latency means that the system has to wait for something to happen, which inevitably means that the CPU might also be waiting/not doing useful work. Examples of latency are disk and device I/O. For example, what happens if we read 1 byte off the disk for each disk read? What about if we only print 1 character to the screen at a time? What happens to utilization when we buffer I/O and print, say, 80 characters to the screen? Why do we read pages from the disk, rather than bytes?

Memory as a Resource

Think about Asst 3, which gives you a very good idea about how to evaluate memory utilization in a system. You can think about memory utilization both for physical memory and virtual memory. Let's go through each step in Asst 3 and discuss how the changes you make to the memory system change utilization for phys/virtual memory.

Before Step 2: You're mapping virtual pages directly to physical pages. What's the utilization of virtual memory for each process? What is the utilization of physical memory? Note that every process has the same memory space-- which means your utilization numbers should be the same for all processes. You can assume some numbers: 100 virtual pages in each process space and 150 physical pages, and the processes are allocating memory at the pace described in Asst 3.

After Step 2: What does step 2 do for processes? As a reminder, step 2 allows you to pick any free physical page to map to a virtual page. However, processes still are not isolated. What happens to physical memory utilization? Virtual memory?

After step 3: Step 3 implements process isolation, so process A and B can both have virtual page 0x1000 allocated. Does this make virtual memory utilization worse or better?

After step 4: Step 4 increases the heap size, which means that each process can allocate many more pages than before. What does this do to physical memory utilization? What about virtual memory utilization?

After step 6: Step 6 allows forked processes to share read-only memory with their parent processes. This is a utilization optimization, because it collapses shared pages into a single page; otherwise, you would have 2 (or more) physical pages that have the same contents, but are repeated throughout physical memory; thus it increases the efficiency and utilization of physical memory.