Overview:

Handling full TLB (vm_fault) - Used the existing code for vm_fault except when there are no more free tlb entries tlb_random is called to replace a random tlb entry.

Read-Only Text Segments - For each address space section (text and data) there is a new field indicating whether it is writable or not. When as_define_region in addrspace.c is called a writable flag is passed in and this is used (when load_elf is called) to determine whether the address space segment should be writable or not. When vm_fault is called and the address space segment has the writable flag set then we set the lower entry address with the TLBLO_DIRTY and TLBLO_VALID flag before writing it into the tlb. If the writable flag is not set then TLBLO_DIRTY is excluded. This was running into issues where the address space was read only but needed to be loaded with data during load _elf. So another flag was created to only set sections as read only in the tlb during a vm_fault call after as_complete_load was called for the that section.

Managing Physical Memory - A single page table is used to manage the all memory in ram. When the virtual memory system bootstraps the total amount of memory available is used to calculate the number of pages available in memory and the size of the page table. The page table is an array of page table entries in an allotted section of the memory. Each entry in the page table contains a field for a physical address, its state, and a pointer to another page. Each page in the page table maps to a physical frame in memory after the alloted space in memory for the page table. When allocating memory with alloc_kpages n contiguos pages in the page table are found and are all linked together and then the virtual address of the first frame in memory is calculated using the phyical address field of that first page table entry. When freeing frames in memory we search through the page table for the virtual address passed in by converting each physical address held in the page table to a virtual address. When the page is found in the page table we follow the linked list of pages and set each page as unused and zero out the physical frame in memory that it maps to. This approach still allows processes to have pages allocated in different regions of memory however whenever more than 1 page is allocated at a time (an array, pointers, large object) it has to be contiguos. So a process may have many objects across different frames in memory but any single object spanning multiple frames would have to be contiguous (did not know how to split it and still appear contiguous, maybe pointers). The addrspace code is mostly the same as the code from dumbvm with some modifications. There are still two sections of memory (one for code and one for data) along with a stack for each address space. The new alloc_kpages allow the sections to be in different regions of memory and the new free method will destroy and zero out those memory regions when the address space is destroyed in as_destroy. The same number of stackpages are used for each address space's stack. However, now the entire memory heap is available for each process.

Changes:

Added a few new semaphores to control the number of processes that can exist in the system at a time and how many times runprogram can be running at any given time. This was to help programs run sequentially without overflowing the memory. I also had to wrap some previous system calls with interrupts off due to some weird memory bugs.

Pros/cons:

Currently there's no access control for which process can free which page frame in memory. I thought about creating a pagetable for each process that would contain information about the pages it has allocated and using that to allow or deny the ability to free a frame in memory. Not having access control for freeing memory is a huge liability for system stability and security.

There are only two sections of memory and a stack defined for each address space. This is a limitation of the system and I considered creating an array for each address space that would hold segments where the segments could be a struct with some information about them. Using the predefined system where its expected that only two sections of memory and a stack would be used did make creating the vm system a lot more simple and potentially saves a little bit of memory.

Issues:

Making segments read only only after the segments were already loaded into tlb. The test programs were failing before this change was made. The fix was to make a segment read only only after as_complete_load was called for the address space.

Kernel stack overflow and memory corruption issues. Some of these issues were fixed by turning interrupts off and most were unavoidable by modiying code but instead were fixed by increasing the memory. I think my current implementation uses a lot of kernel stack memory especially when multiple processes are created as a lock and condition variable are created and stored in a global array for each new process along with each new process being stored in a global process table. Some of the test programs in uw-testbin cannot run unless more memory is used. I think some better memory management besides allocating and freeing memory is needed as the kernel stack seems to grow pretty quickly.

Some tests like uw-testbin/hogparty or testbin/forktest will fail if too many calls to them are stringed together in a short amount of time with smaller ram sizes. This happens because the system takes a second or two to clean up resources between each program run so if many programs are run in a short period of time the kernel stack may overflow the memory and crash the system especially for smaller ram sizes. This could be fixed by adding more ram or accounting for and limiting the kernel stack usage.

Alternatives:
A nested page table where each process has its own page table and the kernel page table has entries for those process' page tables. This was too complex for me to do in the amount of time I had and I thought I should try to make the vm system function in a as simple as possible matter. Hence, the singe global page table, infinite heap space for each proccess, no access control for freeing memory, and not accounting for the kernel stack usage at all.

I considered each process having a list of it's allocated pages to use as a reference when checking if it can free a frame in memory. Again I skipped this for the sake of finishing other requirements on time.