Section 8: Wait/Exit, Address Translation

October 7, 2015

Contents

1 Wait and Exit

This problem is designed to help you with implementing wait and exit in your project. Recall that wait suspends execution of the parent process until the child process specified by the parameter id exits, upon which it returns the exit code of the child process. In Pintos, there is a 1:1 mapping between processes and threads.

1.1 Thinking about what you need to do

"wait" requires communication between a process and its children, usually implemented through shared data. The shared data might be added to struct thread, but many solutions separate it into a separate structure. At least the following must be shared between a parent and each of its children:

- Child's exit status, so that "wait" can return it.
- Child's thread id, for "wait" to compare against its argument.
- A way for the parent to block until the child dies (usually a semaphore).
- A way for the parent and child to tell whether the other is already dead, in a race-free fashion (to ensure that their shared data can be freed).

1.2 Code

Data structures to add to thread.h for waiting logic:

Implement wait:

Implement exit:

2 Vocabulary

- **Exit syscall** - A computer process terminates its execution by making an exit system call. An exit in a multithreading environment means that a thread of execution has stopped running. For resource management, the operating system reclaims resources (memory, files, etc.) that were used by the process. The process is said to be a dead process after it terminates.
- **Wait syscall** - A process may wait on another process to complete its execution. In most systems, a parent process can create an independently executing child process. The parent process may then issue a wait system call, which suspends the execution of the parent process while the child executes. When the child process terminates, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution.
- **Zombie processes** - When a child process terminates, it becomes a zombie process, and continues to exist as an entry in the system process table even though it is no longer an actively executing program. Under normal operation it will typically be immediately waited on by its parent, and then reaped by the system, reclaiming the resource (the process table entry). If a child is not waited on by its parent, it continues to consume this resource indefinitely, and thus is a resource leak. Such situations are typically handled with a special "reaper" process that locates zombies and retrieves their exit status, allowing the operating system to then deallocate their resources. In project 2, if you're mallocing some shared memory for communication between a parent and child, make sure you free this memory after a process is completely dereferenced (i.e. parent+children are all finished executing)
- **Virtual Memory** - Virtual Memory is a memory management technique in which every process operates in its own address space, under the assumption that it has the entire address space to itself. A virtual address requires translation into a physical address to actually access the system's memory.
- **Memory Management Unit** - The memory management unit (MMU) is responsible for translating a process' virtual addresses into the corresponding physical address for accessing physical memory. It does all the calculation associating with mapping virtual address to physical addresses, and then populates the address translation structures.
- **Address Translation Structures** - There are two kinds you learned about in lecture: segmentation and page tables. Segments are linearly addressed chunks of memory that typically contain logically-related information, such as program code, data, stack of a single process. They are of the form (s,i) where memory addresses must be within an offset of i from base segment s. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware or more specifically to the RAM.
- **Inverted Page Table** - The inverted page table scheme uses a page table that contains an entry for each phiscial frame, not for each logical page. This ensures that the table occupies a fixed fraction of memory. The size is proportional to physical memory, not the virtual address space. The inverted page table is a global structure – there is only one in the entire system. It stores reverse mappings for all processes. Each entry in the inverted table contains has a tag containing the task id and the virtual address for each page. These mappings are usually stored in associative memory (remember fully associative caches from 61C?). Associatively addressed memory compares input search data (tag) against a table of stored data, and returns the address of matching data. They can also use actual hash maps.

- **translation lookaside buffer (TLB) -** A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. It stores virtual address to physical address mappings, so that the MMU can store recently used address mappings instead of having to retrieve them mutliple times through page table accesses.

3 Problems

3.1 Conceptual Questions

If the physical memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

If the physical memory size (in bytes) is doubled, how does the number of entries in the page map change?

If the virtual memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

If the virtual memory size (in bytes) is doubled, how does the number of entries in the page map change?

If the page size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

If the page size (in bytes) is doubled, how does the number of entries in the page map change?

The following table shows the first 8 entries in the page map. Recall that the valid bit is 1 if the page is resident in physical memory and 0 if the page is on disk or hasn't been allocated.

Valid Bit	Physical Page
0	7
1	9
0	3
1	2
1	5
0	5
0	4
1	1

If there are 1024 bytes per page, what is the physical address corresponding to the hexadecimal virtual address 0xF74?

3.2 Address Translation

Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page?

List the fields of a Page Table Entry (PTE) in your scheme.

Without a cache or TLB, how many memory operations are required to read or write a single 32-bit word?

With a TLB, how many memory operations can this be reduced to? Best-case scenario? Worst-case scenario?

The pagemap is moved to main memory and accessed via a TLB. Each main memory access takes 50 ns and each TLB access takes 10 ns. Each virtual memory access involves:
- mapping VPN to PPN using TLB (10 ns)
- if TLB miss: mapping VPN to PPN using page map in main memory (50 ns)
- accessing main memory at appropriate physical address (50 ns)

Assuming no page faults (i.e. all virtual memory is resident) what TLB hit rate is required for an average virtual memory access time of 61ns.

Assuming a TLB hit rate of .50, how does the average virtual memory access time of this scenario compare to no TLB?

3.3 Inverted Page Tables

Why IPTs? Consider the following case:
- 64-bit virtual address space
- 4 KB page size
- 512 MB physical memory

How much space (memory) needed for a single level page table? Hint: how many entries are there? 1 per virtual page. What is the size of a page table entry? access control bits + physical page .

How about multi level page tables? Do they serve us any better here?

What is the number of levels needed to ensure that any page table requires only a single page (4 KB)?

Linear Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

Hashed Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

3.4 Page Fault Handling for Pages Only On Disk

The page table maps vpn to ppn, but what if the page is not in main memory and only on disk? Think about structures/bits you might need to add to the page table/OS to account for this. Write pseudocode for a page fault handler to handle this.