**Operating System HW1**

(1) On all modern computers, at least part of the interrupt handlers are written in assembly language. Why?

Because when an interrupt occur, there are corresponding actions have to happen. Those actions cannot be written by high level languages.

(2) If a multi-threaded process forks, a problem occurs if the child gets copies of all the parents threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are writing for keyboard input, on in each process. Does this problem ever occur in single-threaded processes?

No, the read() call will make the calling thread block until the keyboard inputs arrive, in the multithreaded case, one thread in parent process is blocked because of read() call, then another thread call fork(), the child process will be created and copy the parent states, which means the read() call will also the copied. when the keyboard inputs arrive, from the view of OS, only one thread will be unblocked, either read() in parent or child.

while in single thread case, there is only one execution unit, no matter at what time there are only one read() call, that’s why the single-threaded process will not have such problem.

(3) Why would a user thread ever voluntarily give up the CPU by calling thread yield? After all, since there is no periodic clock interrupt, it may never get the CPU back.

Threads in a process cooperate. They are not hostile to one another. If yielding is needed for the good of the application, then a thread will yield. After all, it is usually the same programmer who writes the code for all the threads in one process.



(4) Consider a process that continuously reads in a chunk of data from disk (1MB say), does a heavy computation over that 1MB and then reads in the next chunk (adinfinitum). How would this Multi-Level Feedback queues handle this process?

Since this process will take a lot of CPU time and is also I/o-bound, when it is dealing with I/O-bound work, it will be moved to higher priority queues, and when it do the computation work, it will be moved to low-priority queue. However, if it stays in a lower-priority queue for a long time, it may be moved to a higher priority-queue.

(5)

1. user-level threading
2. no-threading
3. hybrid threading

(6) Discuss why the solution in which every philosopher takes and locks the fork on there left and then tries to grab the fork on their right fails.

Suppose every philosopher takes and locks the fork on the left at the same time, then when they tries to grab the fork on the right, they will find there is no fork available, deadlock occurs.

(7)

1. Describe the purpose for each of the global variables below:

i) reindeerSem:

Controlling the synchronization and mutex between Santa() and reindeer(). As we can see in the code, at the beginning, each reindeer will execute reindeer.P() and then get blocked, and when the Santa() wake up, the Santa() will call reindeer.V() to wake up reindeer, later the getHiched() is able to execute.

ii) santaSem:

Controlling the synchronization and mutex between Santa() and reindeer(). At the beginning, the Santa() will get blocked because of santaSem.P(). When the reindeer number equals to 9, then the reindeer() will call santaSem.V() to wake the Santa up.

iii) mutex:

“mutex” ensure only one process can enter the critical section at each time. This variable initialized as 1, because at first there must be one process is able to get into the critical section.

iv) reindeer:

This variable indicated the number of reindeers. Each reindeer process will revise this variable when it get into the critical section, it this variable equals 9 which means the ninth reindeer arrives.

b) Write the elf portion of the synchronization routine.

// there are N elves.

void elf(){

elfTex.P();

mutex.P();

elves+=1;

if(elves == 3){

santaSem.V();

}

else{

elfTex.V();

}

mutex.V();

getHelp();

mutex.P();

elves -=1;

if(elves == 0) elfTex.V();

mutex.V();

}

(8) Given a 48-bit addressable CPU, a page size of 8KB, a page entry size of 8 bytes, and a ram size of 4GB answer the following questions. You can leave your answers in the form 2x (where you would give x).

a) How big is the (maximum) virtual address space?

2^48 ram size

b) How many entries are there in the page table (assuming the maximum virtual address space size)?

Page Size = 8KB = 2^13B

Offset = 13 bit

number of page entries = 2^(48-13) = 2^35

c) How large (in bytes) would the page table be?

page table size = number of page entries \* page entry size = 2^35 \* 8(bytes) = 2^38byte

d) How large (in bytes) would an inverted page table be?

number of pages = 2^48/2^13 = 2^35

number of bits required to identify each page = 35 bits (5 Bytes)

number of frames = 2^32/2^13 = 2^19

inverted page table size = number of frames \* number of bits required to identify each page = 2^19 \* 5 Bytes = 2560 KB

e) Using a two level multi-level page table where each second- level page table would be a page-size large, what would an address look like? (i.e. How many bits for offset, 2nd level and top level fields?)

13 bits for offset because page size is still 8KB.

13 bits for 2nd level page table because the 2nd level page table would be a page-size large.

Then we have 48 -13-13= 22 bits to address top level field.

f) Explain how translation from a virtual address to a physical address occurs using a multi-level page table.

Take a two-level page table as an example:

the virtual address can be expressed as:

| P1 | P2 | offset |

The P1 is used to index the 2nd level page tables. From P1, we can get which one among 2nd level page tables we should look at.

The P2 is the offset of the 2nd level page table. From P2, we can get which page we should look at.

The offset if used to get the which PTE we should access in one page table;

Then we can get physical frame number through the PTE, and the physical address equals to: physical frame number with offset.

(9)

(a) FIFO

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sequence | Frame 1 | Frame 2 | Frame 3 | Frame 4 | Page Fault |
| read P0 | P0 |  |  |  | Y |
| write P2 | P0 | P2 |  |  | Y |
| read P3 | P0 | P2 | P3 |  | Y |
| write P4 | P0 | P2 | P3 | P4 | Y |
| read P5 | P5 | P2 | P3 | P4 | Y |
| write P3 | P5 | P2 | P3 | P4 | N |
| read P0 | P5 | P0 | P3 | P4 | Y |
| write P2 | P5 | P0 | P2 | P4 | Y |

Number of page fault: 7

(b) optimize:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sequence | Frame 1 | Frame 2 | Frame 3 | Frame 4 | Page Fault |
| read P0 | P0 |  |  |  | Y |
| write P2 | P0 | P2 |  |  | Y |
| read P3 | P0 | P2 | P3 |  | Y |
| write P4 | P0 | P2 | P3 | P4 | Y |
| read P5 | P0 | P2 | P3 | P5 | Y |
| write P3 | P0 | P2 | P3 | P5 | N |
| read P0 | P0 | P2 | P3 | P5 | N |
| write P2 | P0 | P2 | P3 | P5 | N |

Number of page fault: 5

(c) LRU

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sequence | Frame 1 | Frame 2 | Frame 3 | Frame 4 | Page Fault |
| read P0 | P0 |  |  |  | Y |
| write P2 | P0 | P2 |  |  | Y |
| read P3 | P0 | P2 | P3 |  | Y |
| write P4 | P0 | P2 | P3 | P4 | Y |
| read P5 | P5 | P2 | P3 | P4 | Y |
| write P3 | P5 | P2 | P3 | P4 | N |
| read P0 | P5 | P0 | P3 | P4 | Y |
| write P2 | P5 | P0 | P3 | P2 | Y |

Number of page fault: 7

(10)

a, How many addresses fit in an i-node?

5\*32bytes +

b, What is the maximum file-size?

c, How many (maximally-sized) files can fit on the disk?

d, Use the example in Figure 2 to convert the following file offsets (in bytes) to physical block addresses:

100, 301, 633