Homework 1

Mengxiang Jiang
CSEN 5322 Operating Systems

October 23, 2022

Problem 1. Suppose your computer has 512MB of memory. The loaded size of the operating system is 128MB. Now you have 100 programs to run and each of them would take 128MB memory.

a) With an average 70% of I/O wait, what is the current CPU utilization?
Since the OS uses 128MB of the 512MB memory, 384MB is left for the programs.
Since each program uses 128MB, $\frac{384}{128} = 3$ programs can be ran at a time.
CPU utilization $= 1 - p^n$ where p is the probability of I/O and n is the degree of multiprogramming. So CPU utilization $= 1 - 0.7^3 = 0.657$ or 65.7% utilization.

b) Minimum, how many of 512MB blocks of memory will you add to your computer to have the CPU utilization just exceeding 98%?
We need to first solve for n given the CPU utilization is 98%.
$0.98 = 1 - 0.7^n \rightarrow 0.7^n = 0.02 \rightarrow n \log 0.7 = \log 0.02 \rightarrow n = \frac{\log 0.02}{\log 0.7} \approx 10.968$
Since we need to exceed 98%, we also need n to be greater than 10.968, so 11 works.
Each additional 512MB increases n by 4 ($\frac{512}{128} = 4$), we start with $n = 3$,
so we need to increase n by $8 \rightarrow$ we need 2 additional 512MB blocks of memory.

Problem 2. a)What is Direct Memory Access (DMA)?
A memory access scheme that allows data access between I/O and memory without constant CPU involvement.

b)why is it important in a modern computer system?
Without DMA, the CPU would have to be busy during the entire I/O operation and becomes unable to perform other work.

Problem 3. a)What is an interrupt?
When a process does I/O, the process is blocked so the CPU can work on other processes. When the I/O is done, an interrupt happens to have the CPU unblock the process.

b)What is an interrupt request?
A signal to the CPU that an interrupt should happen.

c)How does the operating system deal with interrupt/request?

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt-service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service-procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt-service procedure. It will extract information from the interrupting device controller's registers.
7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
9. Load the new process' registers, including its PSW.
10. Start running the new process.

From page 357 of *Modern Operating Systems* by Tanenbaum and Bos.[5]

Problem 4. For the following "Producer-consumer Problem using Semaphores":

```
semaphore mutex = 1; /* binary semaphore providing mutual exclusion */
semaphore empty = N; /* counting semaphore, counts empty buffer slots */
semaphore full = 0; /* counting semaphore, counts full buffer slots */
void producer(void)
{
    while (TRUE) /* TRUE is the constant 1 */
    {
        int item = produce item( ); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        insert_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}

void consumer(void)
{
    while (TRUE) /* infinite loop */
    {
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        int item = remove item( ); /* take item from buffer */
        up(&mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        consume item(item); /* do something with the item */
    }
}
```

a) Will there be any deadlock if the order of the downs were switched, for the producer or for the consumer individually as well as together (so, 3 combinations)? Explain.

Case 1: Suppose that the two *downs* in the producer's code were reversed in order, so *mutex* was decremented before *empty* instead of after it. If the buffer were completely full, the producer would block, with *mutex* set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a *down* on *mutex*, now 0, and block too. Both processes would stay blocked forever, deadlock.

Case 2: Suppose that the two *downs* in the consumer's code were reversed in order, so *mutex* was decremented before $full$ instead of after it. If the buffer were completely empty, the consumer would block, with *mutex* set to 0. Consequently, the next time the producer tried to access the buffer, it would do a *down* on *mutex*, now 0, and block too. Both processes would stay blocked forever, deadlock.

Case 3: Suppose both the two *downs* in the producer's code and consumer's code were reversed in order, so *mutex* was decremented before *empty* for the producer and *mutex* was decremented before $full$ for the consumer. If the buffer were completely full, it would deadlock as in Case 1. If the buffer were completely empty, it would deadlock as in Case 2.

Problem 5. a)What does PSW (Program Status Word) consist of?
This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits.

b)Starting from 8088 or 8086 processor, show the chronological additions and improvements to the PSW for at least two next-generation computers. Also, explain why the additions and improvements are needed?
In the 8085, the PSW or flags register used 5 bits, namely Carry(C), Auxiliary carry(A), Sign(S), Parity(P), and Zero(Z).
C stores the final carry of the addition of two 8 bit numbers.
A stores the intermediate carry of the addition of two 8 bit numbers (the first 4 bits).
S stores whether the result of an operation is negative.
P stores whether the result of an operation has an even number of 1's in it.
Z stores whether the result of an operation is zero.[3]

In the 8086/8088, 9 bits are used, including the 5 bits of the 8085. The 4 additional bits are Overflow (O), Directional (D), Interrupt(I), Trap(T).
O stores if the result of an operation overflows.
D stores the direction to access string data (1 from high to low, 0 from low to high).
I is set if interrupt handling is enabled (if 0 will ignore all interrupt requests).
T is set for debugging (if 1, CPU automatically interrupts after each instruction).[2]

In the 80286 and above, additional bits used include the Input/Output Privilege Level(IOPL), Nested Task(NT), Resume(RF), Virtual Mode (VM), and Alignment Check(AC).
IOPL stores the I/O privilege level of the current task.
NT indicates that the current task is nested within another task.
RF allows turning off certain exceptions while debugging code.
VM simulates 8086 behavior.
AC used to detect unaligned memory operations.[1]

Problem 6. For the following Peterson's solution from your textbook, for the *enter_region* called by process '0' and process '1', show all possible combinations, interleaving of the execution paths, focusing on testing the 'while-loop'. Will there be any violation of mutual execution? Explain showing all possible cases.

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */
int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0 (FALSE) */
void enter_region(int process);       /* process is 0 or 1 */
{
    int other;                        /* number of the other process */
    other = 1 - process;              /* the opposite of process */
    interested[process] = TRUE;       /* show that you are interested */
    turn = process;                   /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement
    */ ;
}
void leave_region(int process)        /* process: who is leaving */
{
    interested[process] = FALSE;      /* indicate departure from critical
   region */
}
```

Listing 1: Peterson's solution for achieving mutual exclusion

Case 1: Process 0 is interested while 1 is not. Initially neither process is in its critical region. Process 0 calls *enter_region*, sets $interested[0] = TRUE$ and as sets $turn = 0$, thereby entering the critical region. Now whether or not process 1 becomes interested or not, it will not be able to enter the critical region until process 0 leaves and sets $interested[0] = FALSE$.

Case 2: Process 0 is not interested while 0 is not. Initially neither process is in its critical region. Process 1 calls *enter_region*, sets $interested[1] = TRUE$ and as sets $turn = 1$, thereby entering the critical region. Now whether or not process 0 becomes interested or not, it will not be able to enter the critical region until process 1 leaves and sets $interested[1] = FALSE$.

Case 3: Process 0 and process 1 are both interested at the same time. Both call *enter_region* simultaneously. Both store their process number in *turn*. However, the first store is overwritten by the second. Suppose process 1 overwrites process 0, so *turn* is 1. When both processes come to the *while* statement, process 0 sees that *turn* is 1 and exits the *while* and enters the critical region. Process 1 loops until process 0 leaves and sets $interested[0] = FALSE$. Then process 1 enters the critical region. This is the same if process 0 overwrite process 1, only with process 1 entering the critical region first, then process 0 after it.

Problem 7. Synchronization of cooperating processes can be accomplished by a variety of busy waiting solutions, semaphores, monitors, and message passing. Busy waiting is extremely wasteful of resources, so synchronization is generally implemented with semaphores, monitors or message passing. Describe the advantages and disadvantages of each of the three methods. Also indicate the limitations of each method's applicability (e.g., you would not be able to apply monitors if the programming language that you must use does not have support for monitors).

a) Busy waiting
Advantages: It is the simplest method. It is also the fastest response to I/O, since there is no interrupt handling overhead.
Disadvantages: No other process can be executed even when the CPU is doing nothing but waiting. Therefore extremely wasteful and inefficient use of the CPU.
Limitations: No hardware or software limitations.

b) Semaphores
Advantages: Can be used for both mutual exclusion and synchronization.
Disadvantages: Very difficult to keep track of multiple semaphores, not intuitive and hard to document, and often leads to deadlock when implemented incorrectly.
Limitations: Requires special OS systemcalls to work with semaphores while disabling all interrupts. If multiple CPUs are used, requires additional locking to prevent more than one CPU from accessing a semaphore.

c) Monitors
Advantages: Self documenting, since they make explicit the condition for which a thread is waiting, without having to read through the totality of the code, as would be the case if semaphores were used.[4]
Disadvantages and Limitations: Only available in programming languages that support them.

d) Message Passing
Advantages: More flexible than monitors and semaphores, allows distributed computing.
Disadvantages: A lot of overhead for messages as well as acknowledgment of reception means much slower execution.
Limitations: Requires network access for distributed computing.

References

[1] Gary Burt. All about the flags register. https://redirect.cs.umbc.edu/courses/undergraduate/CMSC211/fall01/burt/tech_help/flags.html, August 2001.

[2] Yash R. Flag register of 8086 microprocessor. https://www.geeksforgeeks.org/flag-register-8086-microprocessor//, May 2018.

[3] Yash R. Flag register in 8085 microprocessor. https://www.geeksforgeeks.org/flag-register-8085-microprocessor/, June 2022.

[4] Emin Gun Sirer. The 12 commandments of synchronization. https://www.cs.cornell.edu/courses/cs4410/2012fa/papers/commandments.pdf, October 2011.

[5] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Boston, MA, 4 edition, 2014.