Fintan Guckian

ITC 262

SID: 11630129

Assignment 2: Short Answers

Task 1

F0 - 0x1020 – Load accumulator from memory location 0x20
F1 - 0x6000 – Load BC register with the value 0
F2 - 0x5029 – Add to AC from memory location 0x29
F3 - 0x7000 – Increment BC register by 1
F4 - 0x8AF2 – Jump to F2 if the contents of the BC register is not equal to 10
F5 - 0x2030 – Store AC to memory location 0x30

Before execution

PC	F0
AC	
IR	
BC	

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 1

PC	F1
AC	0003
IR	1020
BC	

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 2

PC	F2
AC	0003
IR	6000
BC	0

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 3

PC	F3
AC	0007
IR	5029
BC	0

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 4

PC	F4
AC	0007
IR	7000
BC	1

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 5

PC	F2
AC	0007
IR	8AF2
BC	1

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 5

PC	F3
AC	000B
IR	5029
BC	1

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 6

PC	F4
AC	000B
IR	7000
BC	2

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 7

PC	F2
AC	000B
IR	8AF2
BC	2

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After cycle 8

PC	F3
AC	000F
IR	5029
BC	2

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

……
……
……

After pre – penultimate cycle

PC	F4
AC	002B
IR	7000
BC	10

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After penultimate cycle

PC	F5
AC	002B
IR	8AF2
BC	10

Memory	
20	0003
:	
29	0004
30	
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

After execution complete

PC	F6
AC	002B
IR	2030
BC	10

Memory	
20	0003
:	
29	0004
30	002B
:	
F0	1020
F1	6000
F2	5029
F3	7000
F4	8AF2
F5	2030

Task 2

2ai.

Since each job locks up four DVD drives there can only be five jobs on the processor at once.

ii.

A maximum of five DVD drives might be idle if all the jobs are in the 'three DVD drives' part of their job. If all jobs are in the 'four DVD drives' part of the job then there are no idle DVD drives which means the minimum idle drives is 0.

2bi.

An alternative policy could be to allow any job to begin if the system has three available DVD drives. If jobs move into their 'four DVD drives' section and all the DVD drives are busy then the next job can be put into a suspend state while it waits for a DVD drive to become free. Since the '4 DVD drives' part of the job is short it should have priority over newly created jobs waiting for three DVD drives to begin.

ii.

The maximum number of jobs that can be running using this policy is six.

iii.

With this policy there will be at most two idle drives when the six processors are on their 'three DVD drives' part. There will be a minimum of 0 idle drives when two of the jobs move into their 'four DVD drives' part while the rest of the jobs are in their 'three DVD drives' part.

Task 3

'THE' multiprogramming system was a system developed by a team at the Technische Hogeschool Eindhoven in 1968 with the purpose of processing “a continuous flow of user programs as a service to the university”(Dijkstra, page 1). THE was designed to run on a Dutch computer Electrologica X8 which had a processor that consisted of an arithmetic unit, two 27 bit accumulators, an instruction register, a condition register and an index register. The computer had 512 words of RAM (28 bit words) and could be attached to a magnetic drum for secondary storage. It also had 48 I/O channels which could be connected devices like tape readers, tape punchers ans teleprinters.

'THE' was structured into a hierarchy of 6 layers which ranged from layer 0 (responsible for selecting processes to run, handling interrupts and context switches) up to layer 5 (user). Layer 1 allocated memory to processes, layer 2 handled input from the console, layer 3 managed I/O and layer 4 was the layer for user programs. A distinguishing feature of this OS was that it introduced semaphores to block processes waiting on an event from using the processor.

The Michigan Terminal System was a time sharing OS developed at the university of Michigan for the purpose of efficiently giving multiple users access to a computers resources. It was originally designed to sit on the IBM 360/67 mainframe computer and was adapted to suit a range of later

IBM and Amdahl computers. In its initial implementation it could support five users. The system used a resource manager to instantiate and monitor processes, make an assessment of the process and give it a priority, handle I/O requests made by the job and terminate the job upon completion or cancellation. The resource manager also kept a record of the job for a short time after termination. Processor time was allocated to users at a terminal in a round robin scheme. The MTS had system and user modes to protect memory from user programs that might try to write in the resource manager space or in the space of other users.

Amiga OS was a personal computing OS developed by Commodore and released in 1985 as the native operating system of the Commodore Amiga 1000. The Amiga 1000 used a Motorola 68000 processor and had 256KB of RAM. It had a kernel that implemented pre-emptive multitasking which contrasted it with the leading computer of the day, the Macintosh. The pre-emptive multitasking was scheduled in a prioritized round robin scheme. One problem the Amiga kernel had was with memory management. The Amiga kernel did not run in privileged mode. This meant that it could not seal off portions of memory from user programs as effectively as a kernel that runs in privileged mode. As a result user programs sometimes encroached upon the kernel's data structures and changed them. This was known to cause problems with the multi tasking functionality as user program encroachment could prevent the kernel from being able to take the running process off the processor. This sometimes led to a program running indefinitely, the only cure being a system reboot.

Task 4

Interrupts can need several hardware pieces to be in place in order to function. When an interrupt is issued by an external device a signal travels down a pin into an interrupt controller. A controller is a chip that is connected to all devices that can issue interrupts. The controller multiplexes all the external device lines into a single line which is connected to the processor. When the interrupt arrives at the processor it sets a flag to alert the processor that a device is waiting. This flag is realised by using a register on the processor which is checked each time the processor finishes an instruction. The processor can then send an acknowledged signal down the line to the controller and lower the interrupt flag. In order to process an interrupt the processor must save the current processes program counter and program status word. This task requires more special registers on the processor. The processor has another register for disabling interrupts. The processor can then indicate to the OS that there is an interrupt to be handled by giving it the ID of the device that called the interrupt. The ID is derived by the processor from the line of the interrupt controller that the interrupt signal travelled down. Once the interrupt has been handled the processor can restore the process by loading the saved context from the registers it was saved in and continue as if the disruption had never occurred.

Mutual exclusion is a concept aimed at protecting the integrity of processes that communicate either directly or through a shared resource (concurrent processes). When processes share access to a resource such as a variable or an I/O device their execution is complicated. There needs to be a mechanism in place to ensure that the actions of process A do not impact upon the correct functioning of process B when they run concurrently. It is important that a process function in its intended way regardless of weather it is blocked for a long time or runs without hindrance. In order to do this programmers use several methods to provide mutual exclusion, which can be thought of as a way of ensuring that a process leaves a resource in a stable, predictable condition for use by the next process.
Semaphores are one way of implementing mutual exclusion. Semaphores are variables that can be used to communicate the state of a shared device (busy/free) and the condition of a shared resource.

For instance, a binary variable can use 0 to indicate a device is in use and 1 to indicate a device is free. The value can then be altered by processes sharing the device as they access and exit their critical time in the device by calling procedure wait to request access and a signal procedure to relinquish control of the device. Critical time is the part of the code during the execution of which a process cannot loose control of the resource.
Monitors are modules that allow semaphores to be controlled from one location. Monitors can be created to only allow one process to use its procedures at a time. This ensures mutual exclusion. Like semaphores they use variables to communicate between devices. Monitors encapsulate the synchronization of processes by only allowing processes to manipulate the variables through the procedures written in the monitor. This is an advantage over semaphores as individual semaphore wait and signal calls can be hard to trace in larger programs which can lead to a frustrating search for the fault in the code.

References

University of Amsterdam, *The electrologica X1 and X8 computers.* Retrieved From https://ub.fnwi.uva.nl/computermuseum//X1.html

Dijkstra, E.W. (1965) *The structure of THE multiprogramming system* Retrieved From http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD196.PDF

The University of Michigan (1991) *The Michigan Terminal System* https://deepblue.lib.umich.edu/bitstream/handle/2027.42/79598/MTSVol01-TheMichiganTerminalSystem-Nov1991.pdf?sequence=1&isAllowed=y

Shichao's Notes. Chapter 7. Interrupts and Interrupt Handlers. https://notes.shichao.io/lkd/ch7/

Stallings, W. (2012), *Operating Systems: Internals and Design Principals.* New Jersey, USA: Prentice Hall