Logan Zweifel

CS:3620

HW3

1. Sequential)

Processes both take 20 min CPU time + 20 min I/O time -> 40 min each -> 80 min total run time

Parallel)

Assuming 50% I/O wait time -> CPU utilization = 1-.5^2= 0.75/2= 0.375 ->

20 min CPU time / 0.375 CPU utilization = 53.3 min total

2)

Diagram

Description automatically generated

3) Yes, a program can be determined to be likely CPU or I/O bound by analyzing the binary code. For example, if you see a lot of binary code that incorporates elements of reading and writing, that would indicate that a program is I/O bound. On the flip side, a program that incorporates many computations is CPU bound. At runtime, this can be determined as well by comparing the number of processes to the total CPU utilization.

4) a – Round Robin) Assuming time slice of 1 min

Process C first to complete at t= 8 -> D at t=17 -> B at t=23 -> E at t=28 -> A at t=30

8+17+23+28+30 / 5 = 21.2 min avg. turnaround time

b- Priority Scheduling)

B completes at t=6 -> E at t=14 -> A at t=24 -> C at t=26 -> D at t=30

6+14+24+26+30 / 5 = 20 min avg. turnaround time

c – FCFS )

A completes at t=10 -> B at t=16 -> C at t=18 -> D at t=22 -> E at t=30

10+16+18+22+30 / 5 = 19.2 minutes avg turnaround time

D – SJF)

C completes at t=2 -> D at t=6 -> B at t=12 -> E at t=20 -> A at t=30

2+6+12+20+30 / 5 = 14 min

5) a)

# entries in the table = 2^38 (VM size) / 2^14 (Page size from 16KB) = 2^24

Table size = 2^24 \* 4b = 2^26 -> equivalent to 64MB

# frames in physical memory = 2^32 / 2^14 = 2^18

So, Virtual Memory -> 24 bit | 14 bit

Physical Memory -> 18 bit | 14 bit

b)

next level

page table size = 2^26B

# entries at this level of table = 2^26 / 2^14 (16KB) = 2^12

Top level

Page table size = 2^12 \* 4 = 2^14 -> Equivalent to 16KB

# bits for 1st (next) level = 12

# bits for 2nd(top) level = 38-14-12 = 12

Results in

Virtual Memory -> 12 bit | 12 bit | 14 bit

Physical Memory -> 18 bit | 14 bit

6)

.99 prob -> physical address in TLB -> access time of 1 ns

.01 prob ->

.0001 prob-> page fault occurs -> access time = 6ms -> equivalent to 6,000,000 ns

.0099 prob -> no page fault -> access time =100ns

(.99\*1) + (.0001\*6,000,000) + (.0099 \* 100 ns) = 601.98 ns -> ~602ns

7)

2^48 (virtual addresses) / 2^13 (8KB) = 2^35 -> ~34 billion

8)

Fragment A

Nested for loops result in X[i][j] running 4096 times (64^2)

-> (i) is used for rows, (j) for columns -> (i) nested in (j) so more row access than column access

-> each loop iteration (i) accesses new row. -> Page can only hold 2 rows worth of array data (128 words /2 rows -> 128/64 -> 2)

So, 1 execution of the (i) loop results in 64/2 = 32 total page faults.

-> (j) loop runs 64 times and incurs no page faults

-> Total page faults = 32\*64 = 2048

Fragment B

-> (j) nested inside of (i) and used for columns, (i) still used for rows (opposite nesting of Fragment A)

-> At beginning of execution, row 0 & 1 put into memory and cause 1 page fault. Since next step is only changing the column and not the row, no further page faults occur. Next execution, row 1 already in memory so no page fault. This pattern continues for whole execution of fragment B

-> Therefore 64/2 = 32 page faults which is the total # of page faults.