Homework 2

**Due:** May 9, 2022 **Points:** 100

When you do the homework, please put the answers to questions 1-4 on one page, and the answers to each of the others on separate pages. You can save this file and put your answers on it. This will make. using Gradescope to grade the assignment much easier than if you submitted everything without regard to pages.

Remember, you must *justify all your answers*.

# Short-answer

1.( *8 points*) What are the four conditions that are necessary for a deadlock to occur?

The four conditions necessary for a deadlock to occur include:

1. Mutual Exclusion - At least one resource must be held in a non-sharable mode
2. No preemption - Once a process is holding a resource ( i.e. once its request has been granted ), then that resource cannot be taken away from that process until the process voluntarily releases it.
3. Circular Wait - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[ i ] is waiting for P[( i + 1 ) % ( N + 1 ) ].
4. Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

2.( *7 points*) Consider a logical address space of 16 pages of 4096 words each, mapped onto a physical memory of 1024 frames. How many bits are there in the logical address? In the physical address?

Logical address: # of pages \* page size

= 16 \* 4096

= 2^4 \* 2^12

= 2^16

The # of bits for the logical address is 16.

Physical memory: # of frames \* page size

= 1024 \* 4096

= 2^10 \* 2^12

= 2^22

The # of bits for the physical address is 22.

3.( *10 points*) How does the Working Set replacement strategy relate job scheduling to memory management?

The working set of information is the smallest collection of information that must be present in main memory to ensure efficient execution of a program. Since there is such an intimate relation between a process and its working set, memory management and process scheduling must be closely related activities. One cannot take place independently of the other.

As a process is running on OS, the OS needs to make sure the process has enough pages for it to have minimum page faults. Hence, the OS must determine which pages are in the working set and which are not at any given moment in time.

The process can only execute when the entire working set is in memory. On top of this, if a process is executing, a page cannot be removed from its working set. If another frame must be grabbed, the process must first be stopped and swapped. If a process is to be brought into memory, there must be enough frames available to bring in its working set.

Due to these required conditions, the Working Set replacement strategy ties process scheduling and memory management.

# Long Answer Questions

4.( *30 points*) Consider a system with three model airplane building processes and one agent process. Each building process requires a tube of glue, a piece of newspaper, and a model kit to put a model airplane together. One of the processes has tubes of glue, another pieces of newspaper, and the third model kits. The agent has an infinite supply of all three. The agent places two of the ingredients for the model builders on the table. The process who has the remaining ingredient can then build one model airplane. It signals the agent upon completion. The agent then puts out another two of the three ingredients and the cycle repeats.

Write a program to synchronize the agent and the model building processes using monitors. Assume that if a process signals on a condition variable that another process is waiting on, the signaler blocks until the other process either leaves the monitor or block

typedef enum items = {glue , newspaper , model};

builder(item i){

    item my\_item = i;

    item missing\_items [2];

    int self\_completed = 0;

    while !(self\_completed):

        builder();

    // find missing items

    for i in items:

        if builder.my\_item != i: missing\_items.append(i)

    void builder(item missing\_item){

        // ask agent for missing items by blocking on item xcond

        while monitor.needed(missing\_item[0]) and monitor.needed(missing\_item[1]):

            // do nothing

        //once missing items are unblocked and available, build

        //set build\_running to 1 to signal a build is in progress

        monitor.build\_running = 1;

        //build

        //set build\_running back to 0 once build is finished

        monitor.build\_running = 0;

        //increment completed to signal completion

        monitor.completed += 1;

        self\_completed = 1;

        //signal on item xcond to release items

        monitor.take(missing\_item[0]);

        monitor.take(missing\_item[1]);

    }

}

agent{

    // choose 2 items at random

    item chosen\_items[]

    //loop while a process is yet to be completed

    while monitor.completed != 3:

        agent();

    void generate\_items(void){

        chosen\_items = random.choice(items,2);

    }

    void agent(void){

            generate\_items();

            //Dequeue process blocked on chosen items by signalling on chosen items

            monitor.take(chosen\_items[0]);

            monitor.take(chosen\_items[1]);

            while build\_running:

                //do nothing

            //once build is finished, build\_running = 0. Regenerate items

    }

}

monitor{

    //builders array initialized with one of each item

    builder builders = [builder(item[0]), builder(item[1]), builder(item[2])];

    int build\_running = 0;

    int completed = 0;

    //condition variable array for each item the agent puts out

    xcond items\_arr= [item[0], item[1], item[2]];

    entry void needed(item i){

        items\_arr[i].wait()

    }

    entry void take(item i){

        items\_arr[i].signal()

    }

}

void main(void){

    monitor monitor;

    agent agent;

}

5.( *20 points*) Assume that we have a paged memory system with a cache to hold the most active page table entries. It takes 20ns to search the cache. If the page table is normally held in memory, and memory access time is 1*µ*s, what is the effective access time if the hit ratio is 85%? What hit ratio will be necessary to reduce the effective memory access time to 1.1*µ*s?

First, look in the cache. If found, go to the memory location.

This takes 20ns + 1*µ*s = **1.02 *µ*s**

If not in cache, we will have searched the cache entirely. Then we will have to access memory to get the page table and frame, and then access memory once again to get the data.

This takes 20ns + 1*µ*s + 1*µ*s = **2.02 *µ*s**

Thus,

EMAT = 0.85(1.02 *µ*s) + 0.15(2.02 *µ*s) = **1.17 *µ*s**

To reduce EMAT to 1.1*µ*s,

x (1.02 *µ*s) + (1-x) (2.02 *µ*s) = 1.1 *µ*s

**x = 0.92 = 92%**

A hit rate of 92% is needed.

6.( *25 points*) A virtual memory has a page size of 1024 words, 8 virtual pages, and 4 physical page frames. The page table is as follows:

|  |  |
| --- | --- |
| *virtual page* | *page frame* |
| 0 | 3 |
| 1 | 1 |
| 2 | not in main memory |
| 3 | not in main memory |
| 4 | 2 |
| 5 | not in main memory |
| 6 | 0 |
| 7 | not in main memory |

1. Which virtual addresses will cause page faults?

pageSize = 1024

va = pageNum \* pageSize

range = va + pageSize-1

|  |  |  |  |
| --- | --- | --- | --- |
| *virtual page* | *Virtual address* | *Virtual Address Range* | |
| 2 | 2\*1024 = 2048 | 2048 | 3071 |
| 3 | 3\*1024 = 3072 | 3072 | 4095 |
| 5 | 5\*1024 = 5120 | 5120 | 6143 |
| 7 | 7\*1024 = 7168 | 7168 | 8191 |

Since page frames for virtual pages **2-3** are not in main memory, virtual addresses **2048-4095** will page fault

Since page frames for virtual page **5** is not in main memory, virtual addresses **5120-6143** will page fault

Since page frames for virtual page **7** is not in main memory, virtual addresses **7168-8191** will page fault

1. What are the physical addresses for 0, 3728, 1023, 1024, 1025, 7800, and 4096?

frameSize = pageSize = 1024

pageFrame = PageTable[va/p]

offset = va % p

pa = pageFrame \* frameSize + offset

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *virtual address* | *page number* | *page frame* | *offset* | *physical address* |
| 0 | 0 | 3 | 0 | 3072 |
| 3728 | 3 | **-** | 656 | **page fault** |
| 1023 | 0 | 3 | 1023 | 4095 |
| 1024 | 1 | 1 | 0 | 1024 |
| 1025 | 1 | 1 | 1 | 1025 |
| 4096 | 4 | 2 | 0 | 2048 |
| 7800 | 7 | **-** | 632 | **page fault** |