4 problems. 70 points. *Closed book. Closed notes. No electronic device.* *Write your name above.*

1. [20 points] A byte-addressable single-level paging system has 48-bit virtual address, 32-bit physical address, and page size of 16 KB. Page table entries are aligned to 32-bit addresses (i.e., the least significant 5 bits are zero). Each page table entry has the following fields: page number; 1-bit "present" field; 12 bits for protection/accessed/etc.

a. **[7 pts]** Draw the page map table for a process. Show the number of entries, the fields and their sizes.

Solution

Page size $= 2^{14}$ bytes.
Virtual address: [34-bit virtual page number, 14-bit offset in page]
Physical address: [18-bit physical page number, 14-bit offset in page]

	present (1)	phy page # (18)	protection / etc (12)	unused (1)
entry 0				
entry 1				
• •				
entry $2^{34}-1$				

Grading: 1 pt each for virtual address, phsyical address, each of the 4 fields, number of entries.

b. **[6 pts]** The hardware has a TLB of 6 entries managed with LRU replacement. Draw the TLB, showing its fields and their sizes. Indicate which part of the TLB is asociatively searched.

	present (1)	vir page # (34)	phy page # (18)	protection / etc (12)	LRU (3)
entry 0					
• •					
entry 5					

3-bit LRU field is needed because there are six entries.
Present and virtual page # fields are searched associatively.

Grading: 1 pt for each field. 1 pt for associatively searched fields.

c. **[7 pts]** A process is allocated three physical pages, numbered 20, 21, 22. Initially, virtual page 0 is mapped to physical page 20 and no other virtual page is mapped to a physical page. The process makes the sequence of memory accesses shown below in the first column; entry i-R (i-W) means page i read (write). Assume LRU replacement.

In the second column, indicate the virtual pages in physical memory after the access at the left is over.

Give the number of page faults and the number of disk page transfers at the end.

Solution

Access	virtual pages mapped	faults	disk transfers
0-R	0c (clean)	0	0
1-R	1c, 0c	1	1
9-W	9d (dirty), 1c, 0c	1	1
0-W	0d, 9d, 1c	0	0
6-R	6c, 0d, 9d	1	1
1-W	1d, 6c, 0d	1	2
4-R	4c, 1d, 6c	1	2
9-R	9c, 4c, 1d	1	1

So 6 page faults and 8 disk page transfers

Grading: 3 pts for virtual pages mapped, 2 pts for page faults (1 pt if off by 1),
2 pts for disk transfers even if off by 1 (1 pt if off by 2).

2. [20 points] This question concerns adding *medium-term scheduling* to GeekOS. Assume GeekOS with semaphores and demand-paging (i.e., GeekOS with projects 3 and 4) and user programs that make use of both features (i.e., calls semaphores and grows stack pages).

Assume you have a `Freeze()` function that (1) intelligently chooses a a user process in the runnable queue or a wait queue, and (2) moves all its virtual memory into a separate "frozen" file, freeing all its physical pages and paging file pages. Assume you have an `Unfreeze()` function that (1) intelligently chooses a user process in the frozen queue, and (2) restores it.

a. Outline how your `Freeze()` function chooses a process to freeze. What extra information (if any) will GeekOS collect for this purpose.
b. Outline how your `Unfreeze()` function chooses a process to unfreeze. What extra information (if any) will GeekOS collect for this purpose.

Be precise and concise. You'll lose points for rambling.

Solution

Metrics of a process used in deciding whether to freeze/unfreeze the process:

1. Frequency of page faults, i.e., fraction of cpu quanta that end in a page fault. Freezable if high.
2. Number of physical pages allocated to the process. Freezable if high.
3. Frequency of blocked P's, i.e., fraction of cpu quanta that end in blocked P. Freezable if high.
4. Frequency of V's, i.e., number of V's per cpu quanta. Freezable if low.
5. Some way to avoid starvation, e.g.:
   - Accumulated frozen time. Freezable if low.
   - Fifo (or any fair) discipline for frozen queue.

Grading: 18 pts for 1, 2, 3, 4. 2 pts for 5.

3. [20 points] Solve the readers-writers problem without busy waiting, using **semaphores** and no other synchronization construct (no atomic read-modify-write, no disabling interrupts, no PCBs, no wait/wakeup, etc.).

Specifically, given functions `f()` and `g()` that always return, write down functions `cf()` and `cg()` that can be called simultaneously by multiple threads such that:

1. (`cf()` calls `f()` exactly once) and (`cg()` calls `g()` exactly once).
2. The following holds at any time:
   (no thread in `f()` and at most 1 thread in `g()`) or (0 or more threads in `f()` and no thread in `g()`).
3. (every call to `cf` eventually enters `f`) and (every call to `cg` eventually enters `g`)
4. Allow multiple simultaneous calls to `f()`.

Be neat and clear. You lose points if I can't understand your code in a reasonable time.

Solution

The solution is RW5 in the multi-threading note, which was left as an exercise. RW5 is the semaphore-based implementation of RW4. It can be obtained from RW4 just as RW2 was obtained from RW1. Because RW4 is RW1 augmented with `consecR` and `waitingW`, RW5 can be obtained from RW2 similarly. In fact, we can dispense with `waitingW` because `atgW` in RW2 plays the same role. Here is the resulting solution; the modifications to RW2 are tagged by "***".

reads

```
Shared variables:
   int ongF ← 0;          // number of ongoing f calls
   int ongG ← 0;          // number of ongoing g calls
   Semaphore(0) gateF;    // cf threads wait here
   Semaphore(0) gateG;    // cg threads wait here
   int atgF ← 0;          // number of threads waiting on gateF
   int atgG ← 0;          // number of threads waiting on gateG
   Semaphore(1) mutex;    // for atomicity of each await
   int consecF ← 0;     // number of consecutive f calls                        // ***
```

```
cf():
   mutex.P()
   if (not (ongG = 0 and
            (consecF < N or atgG = 0))     // ***
      atgF ++
      mutex.V()
      gateF.P()
      atgF --
   ongF ++
   consecF ++                              // ***
   if (atgF > 0 and
       (consecF < N or waitngG = 0))       // ***
      gateF.V()
   else
      mutex.V()

   f()

   mutex.P()
   ongF --
   if (atgG > 0 and ongF = 0)
      gateG.V()
   else
      mutex.V()
```

```
cg():
   mutex.P()
   if (not ongG = ongF = 0)
      atgG ++
      mutex.V()
      gateG.P()
      atgG --
   ongG ++
   consecF ← 0                             // ***
1: mutex.V()

   g()

2: mutex.P()
   ongG --
3: if (atgG > 0)
      gateG.V()
4: else if (atgF > 0)
      gateF.V()
   else
      mutex.V()
```

Grading: 15 pts for satisfying everything but progress for `g`, e.g., RW2, RW3.
About 5-8 pts for solution that does not satisfy safety requirements.

4. [10 points] Solve problem 3 with two changes: (a) allow multiple simultaneous `g()` calls, and (b) use **awaits** instead of semaphores (no other synchronization construct, no busy waiting).

Specifically, given `f()` and `g()`, obtain `cf()` and `cg()` such that:

1. Same as in problem 3.
2. The following holds at any time:
   (no thread in `f()` and 0 or more threads in `g()`) or (0 or more threads in `f()` and no thread in `g()`).

3–4. Same as in problem 3.

5. Allow multiple simultaneous calls to `g()`.

Solution

This is essentially RW4 applied to both readers and writers.

```
Shared variables:
   int ongF ← 0;     // number of ongoing f calls
   int ongG ← 0;     // number of ongoing g calls
   int consecF ← 0;  // number of consecutive f calls currently
   int consecG ← 0;  // number of consecutive g calls currently
   int waitngF ← 0;  // number of waiting cf threads
   int waitngG ← 0;  // number of waiting cg threads
```

```
cf():
   await (true)
      waitngF ++
   await (ongG = 0 and
         (consecF < N or waitngG = 0)
        )
      ongF ++
      waitngF --
      consecF ++
      consecG ← 0
   f()
   await (true)
      ongF --
```

```
cg():
   await (true)
      waitngG ++
   await (ongF = 0 and
          (consecG < N or wait-
ngF = 0)
         )
      ongG ++
      waitngG --
      consecG ++
      consecF ← 0
   g()
   await (true)
      ongG --
```

Grading: 5 pts for satisfying everything but progress for `f` or for `g`. For example

```
cf():
   await (ongG = 0)
      ongF ++
   f()
   await (true)
      ongF --
```

```
cg():
   await (ongF = 0)
      ongG ++
   g()
   await (true)
      ongG --
```

About 2-3 pts for solution that does not satisfy safety requirements (i.e., allows `f` and `g` ongoing simultaneously).

About 2-3 pts for solution that does not allow multiple ongoing `f` calls or multiple ongoing `g` calls.