CS 4410 Prelim 1

Lev Akabas

TOTAL POINTS

59 / 105

QUESTION 1

Processes And Threads 20 pts

1.1 LIFO scheduler 3 / 3

+ **0 pts** A

✓ + **3 pts** **B**

+ **0 pts** C

+ **0 pts** D

+ **0 pts** 2 answers

1.2 Copy-On-Write symmetry 0 / 3

+ **3 pts** P1's changes will be seen by P2

+ **0 pts** Claim that either P1 or P2 is no longer able to write

+ **0 pts** Claim P2's changes will be visible to P1

✓ + **0 pts** **Other, incorrect answer**

+ **2 pts** Partially unclear explanation

+ **0 pts** No discussion of effect on P2's values

+ **1 pts** Correct answer with incorrect details

1.3 Page table pointer in TCB 3 / 3

✓ + **3 pts** **Threads share address spaces**

+ **3 pts** Threads share page tables

- **1 pts** Extra incorrect information

+ **0 pts** Incorrect answer

+ **1 pts** Correct answer incorrect explanation

1.4 Multilevel Feedback Queue 5 / 5

✓ + **2 pts** **Level 1 - Word; Level 2 - TensorFlow**

+ **0 pts** Level 1 - TF; Level 2 - Word Incorrect priorities

✓ + **3 pts** **Word is IO bound with short cpu bursts, TensorFlow is compute bound with long cpu bursts**

+ **2 pts** Mostly correct explanation, but lack of association between resource-bounding (IO/CPU) with CPU time/quanta used

+ **2 pts** Mostly correct explanation, but lack of direct discussion of the resource Word is bounded by

+ **1 pts** Gave priorities based on use case rather than properties of multilevel-feedback queue / properties of the processes

+ **1 pts** Partially correct explanation, but no focus on CPU/IO boundedness as main influence on priority level

+ **1 pts** Partially correct explanation

+ **0 pts** Correct explanation with incorrect queue levels

+ **0 pts** Incorrect explanation of priorities

1.5 How fork works 4 / 6

+ **1 pts** syscall instruction saves registers/program state (or other correct information about syscall behavior)

✓ + **1 pts** **syscall instruction jumps to kernel (give points if answer makes it clear they're using the kernel)**

✓ + **1 pts** **Mention that new PCB is created**

✓ + **1 pts** **New PT is copy of original (address space is cloned)**

+ **1 pts** copy-on-write bits set in PT

✓ + **1 pts** **syscall returns PID of child in parent and 0 in child**

- **1 pts** Inaccurate info for some step (eg mentioned TCB instead of PCB)

- **1 pts** Some step or detail is out of order

+ **1 pts** Security checks or other correct, important info

QUESTION 2

Synchronization 20 pts

2.1 TBTF Bank (lock ordering) 2 / 10

+ **5 pts** Maintains order on lock acquisition

+ **5 pts** doesn't do anything when a.id == b.id

+ **3 pts** Identified deadlock but did not properly

solve it

✓ + 2 pts **Solved deadlock, broke invariant**

+ 0 pts Completely Incorrect

- 1 pts Extra (obviously incorrect) code

- 2 pts Introduces new lock

- 2 pts No code

+ 0 pts Reorders Vs

2.2 Susie (semaphore broadcast) 10 / 10

✓ + 5 pts **while loop on TAS before while loop to wake threads**

✓ + 3 pts **while loop to wake all threads in wait queue**

✓ + 2 pts **spinlock set to 0 before leaving**

+ 0 pts Nothing

- 1 pts Minor problem/bug with code

- 1 pts Spinlock reversed

- 2 pts Bug in code

- 4 pts Major bug in code

QUESTION 3

Barrier 25 pts

3.1 Naked wait 3 / 5

✓ + 3 pts **Naked wait**

+ 2 pts Breaks locality of reasoning

+ 1 pts You tried

+ 0 pts Nothing or says it is a good use or only mentions bad naming

✓ + 0 pts **Instead of locality of reasoning, talks about MESA semantics and/or needing to check condition on line 4 again**

3.2 Barrier re-entry 5 / 5

✓ + 5 pts **This can deadlock, since the last thread to exit will reset ndone to 0, even though some threads might be waiting. Thus ndone will never again reach N.**

+ 2 pts Does not satisfy liveness property

+ 3 pts Got the deadlock idea correct, but incorrect/inadequate explanation and reasoning

+ 2 pts Got the thread re-entry part correct, but doesn't mention deadlocks

+ 0 pts Incorrect

3.3 Barrier re-entry fix 12 / 15

+ 15 pts Functional and Efficient Solution

✓ + 2 pts **added new conditional variable**

✓ + 5 pts **waits on new CV when nout != 0 or nout != N, as necessary**

✓ + 5 pts **broadcasts to new CV when nout == 0 or nout == N, as necessary**

+ 3 pts slept re-entering thread before incrementing ndone (most efficient implementation)

- 4 pts Incorrectly updates shared state, breaks functionality (e.g. threads sleep forever)

- 4 pts incorrect guard (e.g. nout == N instead of ndone == N)

- 4 pts still allows re-entering threads to pass through barrier while previous round is not finished

- 4 pts incorrect use of CVs (no broadcasts, naked waits)

+ 0 pts Not functional and no new CV created/used properly

- 2 pts used `signal` instead of `broadcast`

- 2 pts used semaphores

- 2 pts Breaks some invariant description (e.g. ndone doesn't represent number of completed threads at some point)

QUESTION 4

4 Scheduling Policies 8 / 16

✓ - 8 pts **Incorrectly specifies RR (quanta use and RR algo wrong)**

Q4: even with assumption, quanta must be 2 unless the process has completely finished; queue-based RR is wrong

QUESTION 5

Address Spaces 24 pts

5.1 User-level thread 4 / 6

✓ - 2 pts **creates a new text page that points to the original or new text frame**

5.2 Line 3 0 / 6

✓ - 6 pts **Does not create new page table (virtual address space)**

stack, data, text of new page table lives in a

new page table for pid 0

5.3 Lines 2, 3, 7 0 / 6

✓ **- 6 pts** Not a solution

5.4 Lines 2, 3, 5 0 / 6

✓ **- 6 pts** No solution or no correct modifications

Prelim 1

CS 4410, Cornell University

Fall 2017, Professors Bracy and Sirer

- Write your NetID at the top of each page.
- Please turn off and stow away all electronic devices. You may not use them for any reason for the entirety of the exam. Do not take them with you if you leave the room temporarily.
- This is a closed book and closed notes examination.
- To receive partial credit you must show your work. Please state any assumptions you make.
- You have 120 minutes to complete 105 points. Use your time accordingly. (Approximately 1 minute per point.)

Problem	Topic	Points
1	Processes And Threads	20
2	Synchronization	20
3	Barriers	25
4	Scheduling Policies	16
5	Address Spaces	24
Total		105

Your Name: Lev Akabas

Your NetID: la286

NetID la286

1. Processes And Threads (20 pts)

a. (3 pts) What is the main drawback of Last-In-First-Out (LIFO) schedulers?

A. They are not fast enough
B. They can starve processes
C. They can have large turnaround times
D. They are not real-time schedulers

Answer:

B

b. (3 pts) When a process P1 forks a new child process P2, both P1 and P2 set the copy-on-write flag to 1 for all pages in their respective page tables. What would happen if P2 sets the copy-on-write flag to 1, but P1 does not?

If P1 changes any variable values before P2 executes, then P2 will no longer have the same data in its pages as P1.

c. (3 pts) Would you expect thread control blocks to contain a pointer to a page table? Why or why not?

No, this information would be contained in the PCB of the process that the thread belongs to.

d. (5 points) Let M be a multilevel feedback queue with two levels 1 and 2, where threads in level 1 are given higher priority. Consider a machine with M as its scheduler that is running two processes:

- TensorFlow, a machine learning application that performs a lot of matrix operations
- Microsoft Word, a document editor

Assume that these two processes are in different levels of M. What levels would you expect the processes to be in? **Explain your answer or you will not get credit.**

I would expect Microsoft Word to be in Level 1 and Tensorflow to be in Level 2 because Microsoft Word is regularly interrupted by I/O when a user types on the keyboard, and due to these I/O bursts Microsoft Word will often go back to the waiting stage and must have priority to begin executing again after getting keyboard input.

e. (6 points) Briefly describe what happens when user code executes a `fork()`. Number and describe each distinct step.

1. The kernel allocates a PCB and memory for the child process
2. The kernel copies the parent's PCB, registers, and stack segments into the child's PCB, registers, and stack
3. The kernel creates a new process ID for the child and sets the child's return value register to 0
4. The kernel sets the parent's return value register to the child's process ID.
5. The kernel places both processes on the run queue

NetID la286

2. Synchronization (20 points)

a. (10 points) TooBigToFail Bank (TBTF) handles millions of transactions everyday for financial institutions. Deep within TBTF's code base is a function `transfer` that moves money between two accounts, whose implementation is shown below:

```
1  def transfer(a, b, amount):
2    print "transferring from Acct#", a.id, " to Acct#", b.id
3    P(a.sema)
4    P(b.sema)
5
6    a.balance -= amount
7    b.balance += amount
8
9    V(a.sema)
10   V(b.sema)
```

`transfer` takes in two objects where the `id` field is the account number, `sema` field is a binary semaphore (initialized to 1) and the `balance` field is the balance for the account. Note that accounts can have negative balances. TBTF engineers report that their systems "occasionally slow down then seemingly come to a halt." Explain why and provide a fix.

transfers

If multiple [redacted] are running, this code could cause a deadlock. For example if transfer(x,y,amount) and transfer(y, x, amount) were running concurrently, the first transfer could be waiting for y.sema and the second transfer could be waiting for x.sema and neither would make progress, since each one has what the other needs.

```
def transfer(a,b, amount):
    print "transferring from Acct#", a.id, "to Acct#", b.id
    P(a.sema)
    a.balance -= amount
    V(a.sema)

    P(b.sema)
    b.balance += amount
    V(b.sema)
```

b. (10 points) Susie likes semaphores but she does not like that they can only wake up one thread from the wait queue at a time. To make Susie happy, implement a function `broadcast` for semaphores that wakes all threads in the wait queue.

Assume that the semaphore object has fields `spinlock`, `queue`, and `count`. Assume also that we have provided the following functions:

- `dequeue(queue)` dequeues the head of a queue
- `test_and_set(lock)` operates on integers. It stores 1 in the lock variable and returns the previous value.
- `thread_start(thread)` wakes up sleeping threads

```
broadcast (sema) {
    while (test_and_set(sema.spinlock) == 1) {
        yield ()
    }
    while (sema.queue.length() > 0) {
        thread_start( dequeue(sema.queue))
    }
    sema.spinlock --
```

3. Barriers (25 points)

a. (5 points) Consider the implementation of a standard synchronization barrier (as defined in class) below. cv is a conditional variable, and N is the number of total threads using the barrier.

```
1 def barrier():
2   with self.lock:
3     ndone += 1
4     if ndone != N:
5       cv.wait()
6     else:
7       ndone = 0
8       cv.broadcast()
```

Is this a good use of conditional variables? Explain why or why not **in detail**. Do not just answer with a phrase.

No, line 4 must replace "if" with while. Otherwise a thread can be woken up and will make progress thrugh the code even if the condition is still not true, since the condition is not checked again.

b. (5 points) Consider the barrier implemented below:

```
1  def barrier():
2    with self.lock:
3      ndone += 1
4      while ndone != N:
5        cv.wait()
6
7      if nout == 0:
8        cv.broadcast()
9
10     nout += 1
11     if nout == N:
12       ndone = 0
13       nout = 0
```

The engineering team reports that this barrier occasionally fails. Briefly explain what the problem is:

In line 12, a thread can set ndone to 0. Suppose some other thread had already re-entered the function and incremented ndone. This increment would be overwritten and the condition on line 4 may never become true.

c. (15 points) Re-write the code snippet above to fix the barrier. You can introduce new conditional variables in your solution, but make it clear that it is a conditional variable by using the canonical names (`wait()`, `signal()`, `broadcast()`) for its operations.

The code snippet has been reproduced below for your convenience.

```
1  def barrier():
2    with self.lock:
3      ndone += 1
4      while ndone != N:
5        cv.wait()
6
7      if nout == 0:
8        cv.broadcast()
9
10     nout += 1
11     if nout == N:
12       ndone = 0
13       nout = 0
```

introducing new condition variable

↓

self.ncv = Condition ("nout is N")

```
def barrier():
    ndone += 1
    if ndone != N:
        while ndone != N:
            cv.wait()
    else:
        nout == 0
        cv.broadcast()

    nout += 1
    if nout != N:
        while nout != N:
            ncv.wait()
    else:
        ndone == 0
        ncv.broadcast()
```

NetID la286

4. Scheduling Policies (16 points)

Show the scheduling order for these processes under 2 policies: First Come First Serve (FCFS), Round-Robin (RR) with timeslice quantum = 2. Assume that context switch overhead is 0 and that both new RR and FCFS processes are added to the tail of the queue.

Process	Arrival Time	Processing Time
P2	1	5
P1	0	1
P6	17	2
P4	4	4
P5	7	6
P3	3	2

* When I called a TA over, the TA said that if a process's arrival time is n, then it will not be able to run until time n+1. Therefore, I'm doing the problem under this assumption.

Time	FCFS	RR (quantum=2)
1	P1	P1
2	P2	P2
3	P2	P2
4	P2	P2
5	P2	P3
6	P2	P3
7	P3	P4
8	P3	P4
9	P4	P5
10	P4	P5
11	P4	P2
12	P4	P2
13	P5	P4
14	P5	P4
15	P5	P5
16	P5	P5
17	P5	P5
18	P5	P5
19	P6	P6
20	P6	P6

5. Address Spaces (24 points)

`RunMe` is running with process ID 523, and contains:

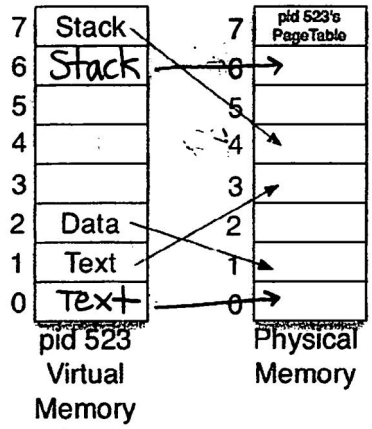
- a 1-page text segment for code
- a 1-page stack segment
- a 1-page data segment for global variables

Part of the code is shown to the right.

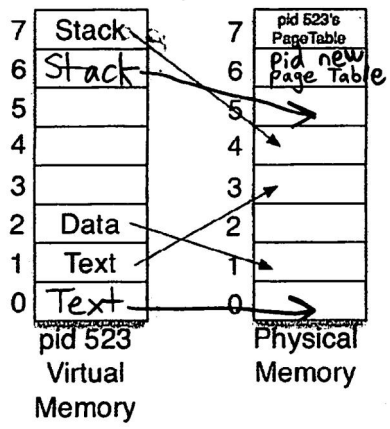
```
1 int someFn() {
2   int x = 6;
3   pid = fork();
4   if (pid==0)
5     exec(RunMe);
6   else
7     x++;  // ld,add,st
```

This question will ask you to update figures showing virtual-to-physical memory mappings. Feel free to add or remove any drawing components as necessary.

a. (6 pts) Suppose `RunMe` spawns a user-level thread (not shown in the code above). Modify the diagram to reflect any changes to the virtual-to-physical mappings.



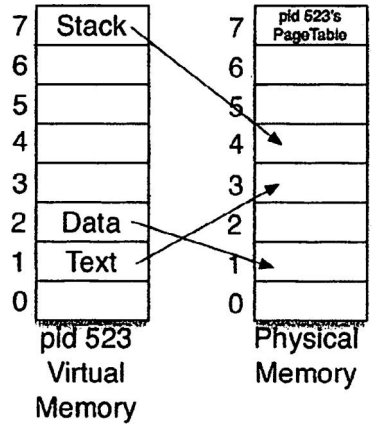
b. (6 pts) Recall that `fork()` copies the virtual-to-physical mappings, marking all frames as *copy-on-write*. Modify the diagram to reflect any changes to the virtual-to-physical mappings after line 3 is executed.



This code is copied from the previous page for your convenience.

```
1 int someFn() {
2   int x = 6;
3   pid = fork();
4   if (pid==0)
5     exec(RunMe);
6   else
7     x++;  // ld,add,st
```

c. (6 pts) Suppose only lines 2, 3, and 7 have executed. Modify the diagram to reflect any changes to the virtual-to-physical mappings.



d. (6 pts) Suppose only lines 2, 3, and 5 have executed. Modify the diagram to reflect any changes to the virtual-to-physical mappings.