**Midterm Practice Questions**

50.005 Computer System Engineering

Materials: Week 1-6 (OS)

*The questions in this document are meant to give you an idea of how we can* *(but not limited to) phrase our longer / analysis-based questions for the midterm examination in Week 8, and to gauge the reasonable level of difficulty. These questions are inspired from past year midterm exam questions. The length of this practice question* ***DOES NOT*** *reflect the length of the midterm exam. In the actual exam, you will have a mixture of simple questions (T/F, MCQ), short (1-2 word) answers, and longer questions. You should also refer to* ***ALL*** *class activities, homework, and bi-weekly quizzes to expose yourself to all kinds of possible question format and enhance your revision.*

1. **[6pts]** Consider the following C program:

int main() {

pid\_t child; // you can treat pid\_t as int

int i, n = 3;

for (i=0; i < n; i++)

child = fork();

}

(a) **[4pts]** Assume that a process P forks a child process Q when P’s value of i is equal to 1.

(i) Immediately after Q returns from the fork, what is Q’s value of i?

**1 [2pts].**

(ii) Will Q itself fork more processes? If so, how many?

**Yes [1pt]; 1 [1pt].**

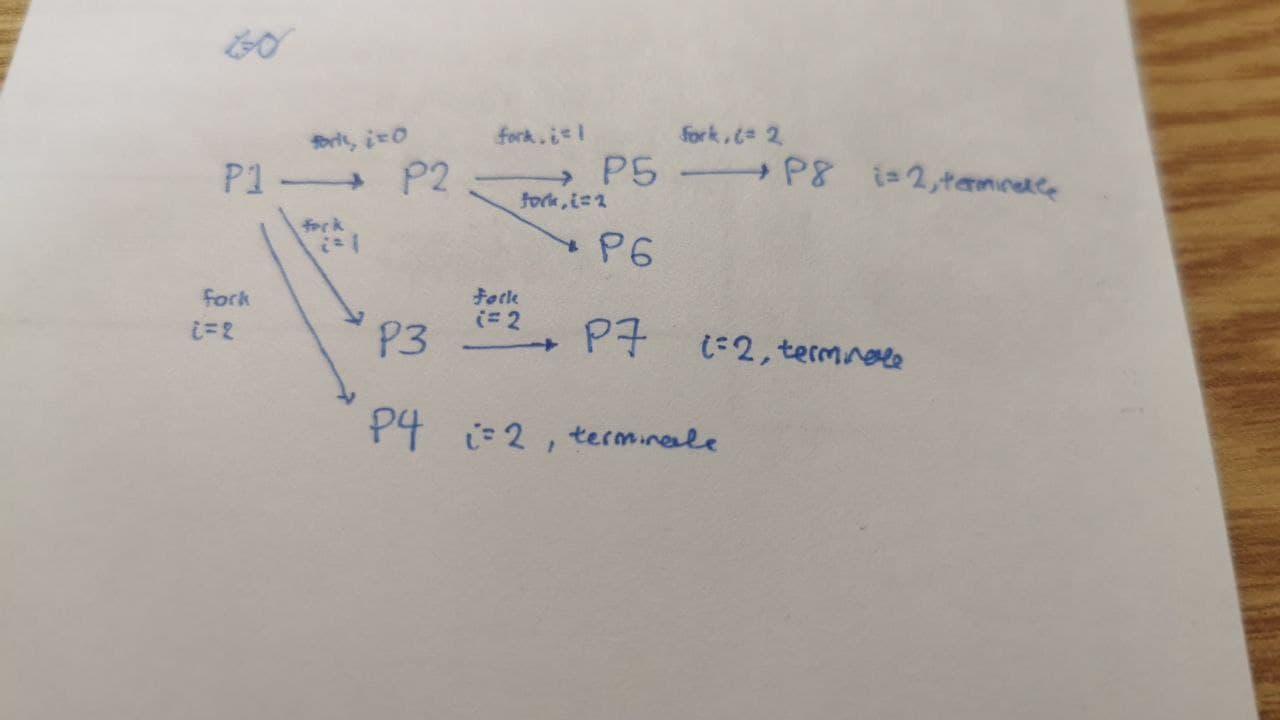
(b) **[2pts]** Draw the process tree of parent-child relationships after executing the shown C program from the very beginning. If you want to name your processes, you can call them P1, P2, etc.

**P1 – P2 – P5 – P8**

**– P6**

**– P3 – P7**

**– P4**

****

2. Consider the producer-consumer problem solved using Java (**unnamed**) condition variables. The insert() method called by the producer to insert a work item into the shared buffer is shown as:

public synchronized void insert(Work item) {

while (count == BUFFER\_SIZE)

try {

wait();

} catch (InterruptedException e) { }

buffer[in] = item;

in = (in + 1) % BUFFER\_SIZE;

count++;

notify();

}

*Note:* The corresponding remove() method by the consumer isn’t shown, but it is **analogous**.

(a**) [2pts]** How does the above solution solve the problem of **mutual exclusion**?

**By using a synchronized method.**

(b) **[2pts]** Assume generally *multiple* producers and consumers. Explain why the solution shown above can be incorrect due to the use of notify().

**Both producers and consumers can be blocked at wait() at the same time. Since notify() selects a random process to wake up, a wrong process may be woken up.**

(c) **[2pts]** Suppose you use notifyAll() instead of notify() in the code shown above. Will the resulting solution be correct? Explain why.

**All the processes blocked at wait() will be woken up, some possibly for the wrong reason. The code recheck the condition upon return from the wait() which means that if the condition no longer holds then the producer/consumer process can return to the wait() set.**

3. **[5pts]** We discussed Peterson’s Algorithm in class for the critical section problem involving two processes i and j:

// Process i’s code is shown – j’s code is symmetric

while (true) {

flag[i] = true;

turn = j;

while (flag[j] && turn == j)

;

// critical section

…

flag[i] = false;

// remainder section

…

}

Assume:

(i) the OS uses **static** priority CPU scheduling so that a runnable higher priority process is **always** selected to run ***before*** a lower priority process,

(ii) Process i runs at priority 10 and Process j runs at priority 20, and

(iii) A process may **block** (sleep) in the remainder section but it will eventually wake up

(iv) The length of the CS is finite

(a) **[3pts]** This system is *problematic*. Demonstrate a scenario for the above system in which a process wants to enter the critical section but will never be able to do so.

**1. i runs and enters CS (still inside it)**

**2. j starts to run and preempts i; gets stuck at inside while loop trying to enter CS**

**3. j wants to enter but it keeps busy waiting forever since it has higher CPU priority**

**[accept alternative solutions and give partial credit where suitable; still accept answer if they think that i has higher CPU priority instead]**

(b) **[2pts]** In your homework, you proved the correctness of Peterson’s Algorithm, including liveness and bounded waiting. What key assumption used in your proof does not hold for the above system?

**The non-zero progress assumption is not true** – the static priority CPU scheduling causes i to starve and get no progress at all**.**

4. **[8pts]** Consider a system with 4 processes and 3 resource types.

Allocation Request Available

A B C A B C A B C

P1 1 0 0 0 1 0 0 0 0

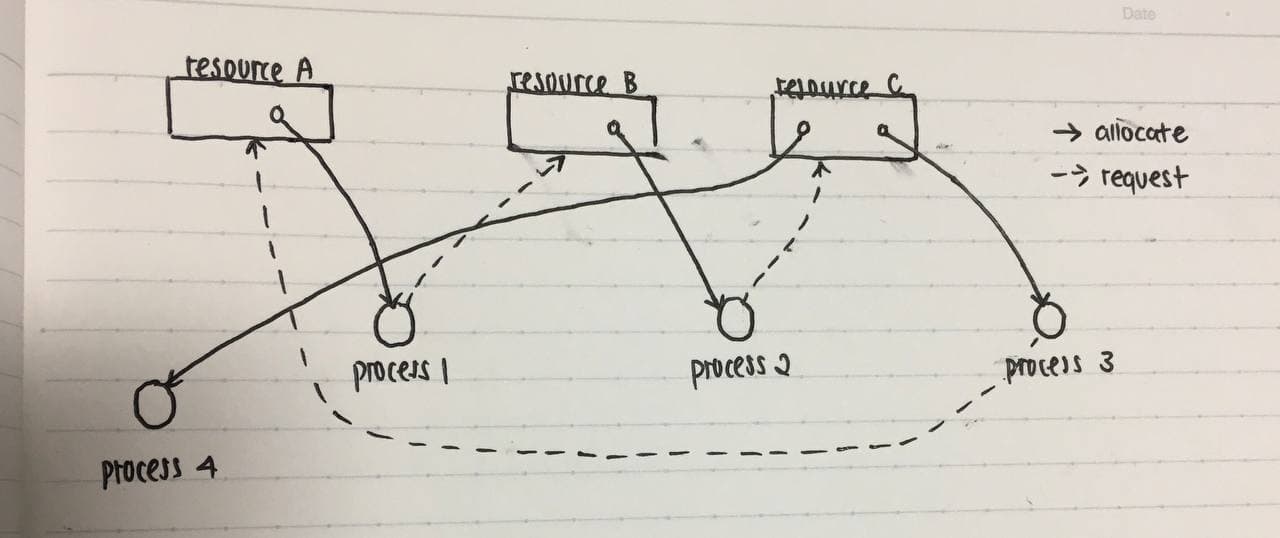
P2 0 1 0 0 0 1

P3 0 0 1 1 0 0

P4 0 0 1 0 0 0

(a) **[3pts]** Draw the resource allocation graph for the above system.

**Surely you know the answer to this question.**



(b) **[3pts]** Is the above system deadlocked? If so, say which processes are involved a deadlocked. If not, give a possible termination sequence. Show your steps.

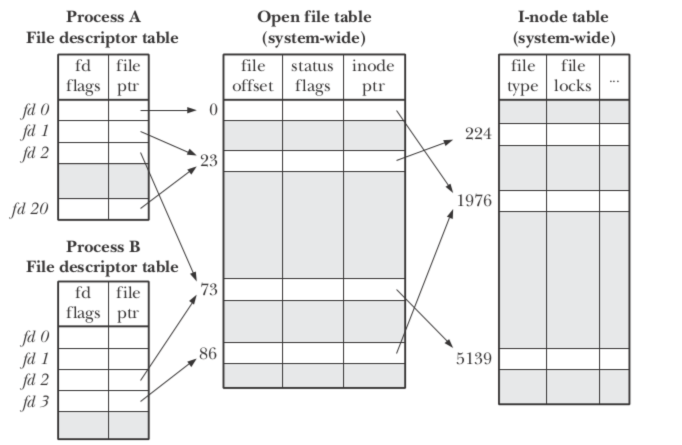
**No [1pt]; P4, P2, P1, P3 is a possible termination sequence of all the processes [1pt]. Showing the step (running deadlock detection algorithm) grants another 1pt.**

(c) **[2pts]** For a general resource allocation graph (i.e., not necessarily the one you draw in Part (a) above), if you determine that the processes are deadlocked, suggest one way to handle the situation.

**We can abort processes involved in the deadlock one by one (and check if deadlock still persists). Continue when there’s no more deadlock [2pts].**

**Alternative answer: we abort all processes and restart [2pts].**

5. **[8pts]** Consider the following snapshot of a Unix file system:



Assume that files 224 and 1976 are text files, containing respectively the sentences “I\*have\*a\*few\*words\*fast\*to\*read”

and “You\*can\*see\*what\*I\*mean\*if\*careful”.

(*NB*: The open and close double quotes are **NOT** part of the sentences.) Initially, all the files are opened for access from the beginning (i.e., file byte offset is 0).

(a) **[4pts]** Give the characters returned after each access in the following *sequence* of accesses (i.e., action ii is taken after action i, etc):

(i) Process A reads 8 characters through file descriptor (fd) 0,

(ii) Process B reads 6 characters through fd 3,

(iii) Process A seeks to offset 6 through fd 20, then reads 7 characters through

fd 20, and

(iv) Process A reads 3 characters through fd 1. *NB*: Each character is one byte

long.

**You\*can\* [1pt]**

**You\*ca [1pt]**

**\*a\*few\* [1pt]**

**wor [1pt]**

(b) [**4pts]** Assume that in a file descriptor (fd) table shown above, a **shaded** area or an empty “file ptr” field corresponds to **unused** entries; e.g.,

(i) entries 3 to 19 in A’s fd table shown are unused, and

(ii) entries 0 and 1 in B’s fd table shown are unused.

Assume that A forks B, and A’s fd table is as shown at the time of the fork. Immediately after B is created, give a **MINIMAL** sequence of file operations (e.g., either open, close, read, write, dup) by B that will result in the fd table shown for B.

For simplicity, you can assume the following definitions of the file operations:

open(<file name>) returns the first originally unused fd as the new fd;

close(<fd>) does not return any values;

read(<fd>, <buffer>, <buffer size>) returns the number of bytes read successfully;

write(<fd>,<buffer>, <buffer size>) returns the number of bytes written successfully;

dup(<fd>) returns the first originally unused fd as the new fd;

In your answer, if the values of any parameters don’t matter, you can use arbitrary plausible values for them, e.g., read(3, buf, 10) without needing to explain or define buf or 10 if they don’t matter.

**open(“whatever.txt”);**

**close(0);**

**close(1);**

**close(20);**

**[The open must be before the close of 0 and close of 1 – deduct 1.5 pts for violating this condition; otherwise, these operations can be reordered. Give partial credit if the sequence of operations results in a partially correct fd table for B, or contains extraneous operations that aren’t needed.]**