

The exam is worth 100 points. There are 8 questions and 7 pages. Answer all questions.

Please be brief, succinct, and to-the-point in your answers.

You will lose points for incorrect statements--even when your answer is correct!

(15 pts) (1) (a) What is an invariant property of a program?

A property that holds throughout the execution of the program—before and after the execution of every atomic instruction.

(b) Define the safety condition for concurrent (or sequential) programs.

Some bad condition, P_{Bad} , never occurs during program execution.

(c) Give an example of a safety condition.

*Division by zero.
Violation of the mutual exclusion property of the CS problem.*

(d) How do you prove that a concurrent (or sequential) program is safe? What type of logic is used for the proof?

*Using Program Logic, locate and prove the Invariant property I .
Let P_{Bad} be the negation of the safety condition.
Prove that $I \rightarrow P_{Bad}$ holds.*

(e) Define the liveness condition for concurrent (or sequential) programs.

Some good condition P_{good} eventually occurs during program execution.

(f) Give an example of a liveness condition.

*Bounded waiting.
Progress.*

(g) How do you prove that a concurrent (or sequential) program satisfies the liveness condition? What type of logic is used for the proof?

Using Temporal Logic, prove that “eventually P_{good} ” holds.

(8 pts) (2) (a) List four pieces of information contained in a PCB (Process Control Block).

Program counter

Registers

Memory Limits

List of Open files.

(b) What does DMA (Direct Memory Access) refers to, in the context of CPU-device interaction?

Data transfer from/to device to/from main memory occurs, once started, without the supervision of the CPU. At the end of the transfer, CPU is interrupted to inform the OS the status of the data transfer.

(c) List two advantages of parallel OSs over centralized OSs.

Increased throughput

Economy of Scale

Increased reliability

(d) Describe how CPU as a resource is protected from intentional or unintentional misuse.

Timer interrupts.

Each process is given a time slice to execute, after which it is interrupted using a timer interrupt, and the CPU is taken away.

(15 pts) (3) Consider the Bakery algorithm given below.

```
Pi: repeat
    choosing[i] := true;
A---→ number[i] := max (number[0], number[1], ..., number[n-1])+1;
    choosing[i] := false;
    for j:=0 to n-1 do begin
        while choosing[j] do skip; ---(*)
        while (number[j], j) < (number[i], i)
            and number[j] ≠ 0 do skip;
    endfor
    Critical Section
    number[i] := 0;
    Remainder Section
until false;
```

Initially, all choosing values are false, and all number values are zeroes.

- (a) Assume that the line (*) is removed. Describe a scenario in which the algorithm does not work (i.e., it does not solve the critical section problem). Briefly, explain what goes wrong and why.

Violation of ME can occur.

Example:

P1: Wants to enter CS. Number[1]=0. Leaves CPU at A, after computing the max() function, but before executing the assignment statement.

P2: Wants to enter CS. Picks number[2] as, say, 250. Goes into CS.

P1: Picks number[1] as 250 also. Goes into CS. Violation of ME!

- (b) Is the Bakery algorithm FIFO (First-In-First-Out) with respect to CS execution? Explain why/why not.

No. Consider the scenario:

P1: Wants to enter CS. Number[1]=0. Leaves CPU at A, after computing the max() function, but before executing the assignment statement.

P2: Wants to enter CS. Goes into CS.

P1: Picks number[1] and goes into its CS.

P2 arrives after P1, but goes into its CS first.

- (c) Does the Bakery algorithm satisfy the bounded waiting property? If no, why not? If yes, what is the bounded waiting value?

BW is satisfied, and the BW bound is (n-1) where n is the number of processes. This happens when all processes pick the same number, and, by luck, the process that came in first is Pn.

(10 pts) (4) Consider a two-process critical section problem algorithm given below.

```
Pi :  repeat
      flag[i]:= true;
      turn:=j;
      while flag[j] do while turn=j do no-op;
      Critical Section
      flag[i] := false;
      Remainder Section
      until false;
```

where flag is a boolean array with two elements, both initialized to False. Is the above solution correct, in the sense that the processes repetitively execute their CS and Remainder section code? Explain why or why not?

Not correct. Both processes can be stuck at the outer while loop.

(15 pts) (5) Assume that you have a sequential programming language of your choice, the concurrent statement (i.e., the parbegin/parend statement), semaphores and wait/signal semaphore primitives. Assume that integer variables A, B, C, D, E and F are already initialized. Write a concurrent program fragment that computes the formula below (without algebraic transformations on the formula) with maximum concurrency.

$$((A * B) + (C * D)) / (E * F)$$

You can use temporary variables to hold intermediate results.

Semaphores a, b, c, d are set to 0 initially.

```
parbegin
  begin X := A * B; signal(a); end;
  begin Y := C * D; signal(b); end;
  begin wait(a); wait(b); Z := X + Y; signal(c); end;
  begin V := E * D; signal(d); end;
  begin wait(c); wait(d); Out := Z / V; end;
parend;
```

(7 pts) (6) Consider the conditional critical region statement “**region v when B do S**” and its implementation with semaphores below.

```

wait(x-mutex);
if not B then begin
    x-count:=x-count + 1;
    signal(x-mutex);
    wait(x-wait);
    while not B do begin
        x-temp:=x-temp + 1;
        if x-temp < x-count then signal(x-wait) else signal(x-mutex);
        wait(x-wait);
    endwhile;
    x-count:=x-count - 1;
endif;
S;
if x-count > 0 then begin x-temp:=0; signal(x-wait) end else signal(x-mutex);

```

Initially, x-mutex semaphore is 1; x-wait semaphore is 0, and integers x-count and x-temp are zeroes. Give a semaphore-base implementation of the other conditional critical region construct we have seen in the class:

```

region v do begin S1; await(B); S2; end;

```

Let A and B denote the if statements before and after S in the above code.

Soln:

```

Wait(x-mutex);
    S1;
A;
    S2;
B;

```

(15 pts) (7) In the assignments, you have given various solutions to the **Cookie Jar Problem**, where Tina, being the older sister, could get a cookie from the cookie jar and eat whenever she wanted to, whereas Judy, to eat a cookie, had to wait for Tina to eat at least two cookies. Fix this unfairness, and implement a solution with strict alternation (using whichever concurrent programming construct--or without any such construct if you wish). That is, Tina and Judy strictly alternate in getting a cookie from the cookie jar, and eating it.

Tina: P₁ Judy: P₂

j = |i - 1| 0 ≤ i, j ≤ 1 Set turn to i initially.

P_i : repeat
 while turn ≠ i do skip;
 GET-COOKIE;
 turn = j;
 EAT-COOKIE;
 DO-SOMETHING;
until False;

-----Alternative-----

semaphores: T, J, initially, one is set to 1, and the other to 0.

Tina:
repeat
 wait(T);
 GET-COOKIE;
 signal(J);
 EAT-COOKIE;
 DO-SOMETHING;
until False;

Judy:
repeat
 wait(J);
 GET-COOKIE;
 signal(T);
 EAT-COOKIE;
 DO-SOMETHING;
until False;

(15 pts) (8) Below is a monitor-based solution to the Savings Account Problem that does not create a list of waiting customers.

type Account = **monitor**

var balance : **real**; needed : **real**;

a, b: **condition**;

procedure Withdrawal (**var** withdraw: **real**)

begin

if (needed > 0) **then** b.wait; -----B

if (balance < withdraw) **then**

begin

needed := withdraw - balance; a.wait;---A

end;

balance := balance - withdraw;

b.signal;

end;

procedure Deposit (**var** dep: **real**)

begin

balance := balance + dep;

if (needed > 0) **then**

if (dep ≥ needed) **then**

begin needed := 0; a.signal; **end**

else needed := needed - dep;

end

begin balance := ...; needed := 0; **end**. //Initialization

Study the solution above, and answer the questions below.

- (a) Does the solution above service the withdrawing customers FIFO? Explain why or why not.

Yes. The first waiting withdrawal customer gets blocked at A. All others get blocked at B in the order of arrival; and, when the first waiting withdrawal customer is serviced, the next FIFO waiting withdrawal customer from B is released.

- (b) Does the solution above service the withdrawing **and** the depositing customers FIFO? Explain why or why not.

No. Depositing customers always get serviced regardless of the waiting customers that have arrived earlier.

- (c) Assume that the balance is \$10. And, the following requests arrive, in order:

1. W1: Withdraw \$20

2. W2: Withdraw \$5

3. D1: Deposit \$10

4. D2: Deposit \$10

- (i) Describe the steady state after events (requests) 1, and 2. That is, where in the code above are the withdrawing processes blocked?

W1 is blocked at A. W2 is blocked at B.

- (ii) Describe the steady state after events (requests) 1, 2, and 3 occur. That is, where in the code above are the withdrawing processes blocked?

W1 withdraws and leaves. W2 gets blocked at A.

- (iii) Describe the steady state after events (requests) 1, 2, 3 and 4 occur. That is, where in the code above are the withdrawing processes blocked?

W2 withdraws and leaves. Balance: \$5.