

The exam is worth 100 points. There are 6 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!

(20 pts) (1) (a) Name the two **concurrent** computation models for communicating processes.

*Shared Memory Model, Message Passing Model.*

(b) State the difference between a sequential programming language and a concurrent programming language.

*A SPL program has a single executing process /single program counter/single line of control.*

*A CPL program has multiple concurrently executing processes/multiple program counters/multiple lines of control.*

(c) Computers are deterministic. And, yet, when we debug a concurrent program, we assume a nondeterministic behavior of processes executing the program. Why?

*A deterministic program always has a deterministic state transition. A nondeterministic program randomly selects its next state. You have always written deterministic programs.*

*Concurrent process execution "seems" random (nondeterministic) because the behavior of the CPU scheduler is unknown, and thus, the process that will execute next is unknown.*

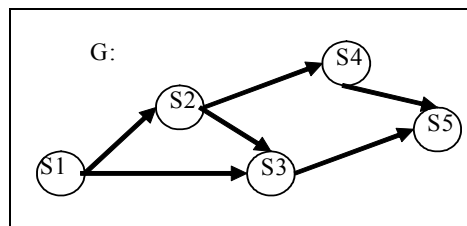
(d) List the concurrent programming language constructs that you have seen in the class.

*Fork/join statements; the concurrent (cobegin/coend) statement;  
test-and-set instruction; swap instruction;  
semaphore primitives wait() and signal();  
critical region statement; conditional critical region statement;  
monitors; condition variable primitives wait.x and signal.x*

(e) Consider a condition variable a in a monitor. Assume that, at the moment, there are no processes waiting on a (that is, there are no blocked processes at an instance of "wait.a"). Will the execution be erroneous if a process inside the monitor executes "signal.a"? What will happen?

*It will not be erroneous; the signal will be ignored.*

(10 pts) (2) Consider the precedence graph G below where nodes represent statements in a sequential programming language, and edges represent the order of execution (i.e., the precedence) among statements.



(a) Can you write a concurrent program with only the concurrent statement (i.e., the parbegin/parend statement, also known as the cobegin/coend statement) that executes the precedence graph G? Yes or no? Explain your answer.

*No. Concurrent Statement can model only single-source/single-sink precedence graphs (possibly with nested single-source/single-sink subgraphs). G is not a single-source/single-sink graph.*

(b) Can you write a concurrent program with only the fork/join statement that executes the precedence graph G? Yes or no? Explain your answer.

*Yes. Fork/Join can model any directed acyclic (precedence) graphs (DAG). G is a DAG.*

(20 pts) (3) Assume we have a bounded buffer B of size n. Consider the following semaphore-based solution to the Producer-Consumer problem where producers “produce” by placing items into B and consumers “consume” by taking items out of B.

<i>Producer:</i>	<i>Consumer:</i>
while True	while True
{ CREATE-ITEM;	{ wait (full);
wait (empty);	wait(mutex);
wait(mutex);	CONSUME; //Remove item from B
PRODUCE; //Add item into B	signal(mutex);
signal(mutex);	signal(empty);
signal(full);	DO-SOMETHING-WITH-ITEM;
}	}

where mutex is a nonbinary semaphore initialized as 1, and full and empty are nonbinary semaphores (that we have seen in the class) initialized as zero and n, respectively.

(a) With mutex as a nonbinary semaphore, the solution above satisfies the three Critical Section Problem (CSP) properties: mutual exclusion (ME), progress (PR), bounded waiting (BW)). Does it also satisfy all three properties when mutex is a binary semaphore? Explain your answers.

ME: *Yes. Whenever a producer or a consumer is in its CS, mutex is zero, and all other producers and consumers are blocked.*

PR: *Single consumer/single producer: No.  
Multiple consumers/multiple producers: No.*

*The processes do not decide among themselves as to which process will drop into its CS; the CPU scheduler does.*

BW: *Single consumer/single producer: Yes. The bound is n. In its attempts to consume (produce), the consumer (producer) can lose out to the producer (consumer) due to the CPU scheduler's choices at most n times.*

*Multiple consumers/multiple producers: No. A consumer may keep waiting at wait(mutex) as other consumers and/or producers keep consuming and producing since the CPU scheduler randomly chooses which process will successfully execute wait(mutex) when the value of mutex is one.*

- (b) When we have one producer and one consumer, the solution given above satisfies the CSP problem. Does it also satisfy the CSP problem when we have  $p$  producers and  $c$  consumers where  $p > 1$  and  $c > 1$ ? With respect to ME, PR, BW, if your answer is “no”, give a scenario illustrating why it will not work. If your answer is “yes”, explain briefly why it will work.

ME: *Yes.*

*Semaphore primitives `wait(mutex)` and `signal(mutex)` serialize the CS executions.*

PR: *Yes.*

*Subject to empty/full semaphore constraints, the producers' and consumers' arrival order at `wait(mutex)` determines which process will enter into its CS. (And, this decision is taken in finite time).*

BW: *Yes.*

*The waiting is FCFS, and is bounded by the size of the waiting list  $L$  of the nonbinary mutex semaphore.*

- (c) What goes wrong if the statement “`wait(empty)`” is removed from the producer code?

*One answer: Producers try to produce when the buffer is full. Assume buffer is full (i.e.,  $full=n$ , and  $empty=0$ ), and no consumer is consuming (i.e., no consumer is in its CS). A producer will execute `wait(mutex)`, enter into its CS, and try to produce when there is no space in the buffer to produce an item.*

- (d) Assume that `full` and `empty` are binary semaphores, and are initialized to zero and 1, respectively. Also assume that we have one producer and one consumer. Explain what would go wrong.

*Only the first element of the buffer will be used by the producer and the consumer; the remaining  $n-1$  elements will be unused (this is really not an erroneous execution—just an inefficient execution).*

(15 pts) (4) In the class, we have seen the **Readers-Writers problem** where **readers starve** in the sense that a stream of writers can arrive and “write” one after another even when a reader has been waiting to read. Consider the following semaphore-based solution.

*Global variables:*

```
nonbinary semaphore mutex, rmutex ; (both initially 1);
nonbinary semaphore wtr, rdr; (both initially set to 0);
int nreaders = 0, nwriters = 0;
boolean Busy = False;
boolean RBlocked = False;
```

*Reader:*

```
while True
{ wait (r-mutex); -----B
  wait (mutex); -----C
  if (nwriters > 0)
    then {RBlocked=True;
          signal(mutex); wait(rdr)} ---A
    else {nreaders++; signal(mutex) };
  signal(r-mutex);
  READ; //CS code
  wait(mutex);
  nreaders--;
  (*) if (nreaders = 0 and nwriters > 0)
    then {Busy = True; signal(wrt)};
    signal(mutex);
    DO-SOMETHING;
}
```

*Writer:*

```
while True
{ wait(mutex);
  nwriters++;
  if ( Busy or nreaders > 0)
    then {signal(mutex); wait(wrt)}
    else { Busy = True; signal(mutex)};
  WRITE; //CS code
  wait(mutex);
  nwriters--;
  Busy = False;
  if (nwriters > 0)
    then {Busy = True; signal(wrt)}
    else if (RBlocked)
      then {RBlocked = False;
            nreaders++; signal(rdr)};
  signal(mutex);
  DO-SOMETHING;
}
```

(a) If mutex and rmutex are binary semaphores, initialized as 1, will the above solution still allow readers read concurrently, writers write mutually exclusively? If no, give a scenario illustrating what goes wrong. If yes, explain why **and** answer whether binary or nonbinary semaphores are better in terms of performance.

*Yes. Wait(mutex) and signal(mutex) (as well as wait(rmutex) and signal(rmutex)) are used only to mutually exclusively execute entry/exit sections. And, for the purposes of mutually exclusive execution, binary and nonbinary semaphores work the same way.*

(b) If mutex and rmutex are binary semaphores, initialized as 1, will there be a case of livelocks? Where?

*Yes; at every wait(mutex) and wait(r-mutex) (except for wait(mutex) at C)*

(c) Assume writer  $W_1$  is in its CS, and three readers  $R_1, R_2, R_3$ , one by one and in the given order, finish their DO-SOMETHING code, and attempt to enter into their CSs. Where will each reader be blocked? What will be the values of variables nreaders, nwriters, Busy, and RBlocked **after** all three readers are blocked?

*R1 at A; R2 and R3 at B.*

*nreaders=0;*

*nwriters=1 (assuming no other writers are waiting to write)*

*Busy=True*

*RBlocked=True*

(d) What goes wrong if the connective **and** in statement (\*) gets replaced by the connective **or**? Give a scenario illustrating the problem.

*One erroneous execution: Assume nwriters=0, nreaders=0, wrt=0. Reader R1 goes through its CS, comes out, sets Busy to True, signals wrt: wrt=1. Reader R2 goes through its CS, comes out, sets Busy to True again, and signals wrt: wrt=2. Now, writer W1 comes, executes wait(wrt) (which sets wrt to 1), and enters into its CS. Writer W2 comes, executes wait(wrt) (which sets wrt to 0), and enters into its CS. ME violation!*

(15 pts) (5) Assume we have the following incorrect solution to the readers-writers problem where readers starve. What goes wrong? Illustrate with a scenario.

*Global variables:*

```
nonbinary semaphore mutex, initially 1;
    wtr, rdr : both initially 0;
int nreaders = 0, nwriters = 0;
boolean Busy = False;
```

*Reader:*

```
while True
{ wait (mutex);
  if (nwriters > 0)
    then {signal(mutex); wait(rdr)}---A
    else {nreaders++; signal(mutex) };
    READ;
    wait(mutex);
    nreaders--;
  if (nreaders = 0 and nwriters > 0)
    then {Busy = True; signal(wrt)};
    signal(mutex);
    DO-SOMETHING;
}
```

*Writer:*

```
while True
{ wait(mutex);
  nwriters++;
  if ( Busy or nreaders > 0)
    then {signal(mutex); wait(wrt)}
    else { Busy = True; signal(mutex)};
  WRITE;
  wait(mutex);
  nwriters--;
  Busy = False;
  if (nwriters > 0)
    then {Busy = True; signal(wrt)}
    else if (nreaders > 0)
      then {nreaders++; signal(rdr)};
  signal(mutex);
  DO-SOMETHING;
}
```

*Multiple errors are possible. One example: readers blocked at wait(wrt):*

*Writer W1 is writing.*

*R1, R2, and R3 sequentially attempt to enter into their CSs, and are blocked at wait(rdr).*

*W1 leaves its CS, and releases R1 into its CS.*

*R2 and R3 are still blocked at A, and not released. The correct behavior should be the release of all three readers into their CSs.*

(20 pts) (6) Consider the Cookie Jar Problem: a cookie jar that never becomes empty is shared by two sisters, Tina and Judy, using the following rule: Judy can get a cookie from the jar only after Tina (being the older sister) gets a cookie in at least two separate occasions, whereas Tina gets a cookie from the jar whenever she wants to. You are to view Tina and Judy as two independently executing processes and the Cookie Jar as a shared resource. Consider the following monitor-based solution to the Cookie Jar problem:

```
type CookieJar = monitor
int TinaCount;
bool Busy;
condition tina, judy;
```

```
procedure entry Tina-OpenJar ( );
begin
    if (Busy) then tina.wait;
    Busy = True;
    TinaCount++;
end;
```

```
procedure entry Tina-CloseJar ( );
begin
    Busy = False;
    if (TinaCount > 1)
    then judy.signal;
end;
```

```
begin TinaCount = 0; Busy = False end;
```

MONITOR USE:

```
Tina:
Var CookieJar-Instance : CookieJar;
While True
{ CookieJar-Instance.Tina-OpenJar ( );
  GET-COOKIE;
  CookieJar-Instance.Tina-CloseJar ( );
  EAT-COOKIE;
  DO-SOMETHING;
}
```

```
procedure entry Judy-OpenJar ( );
begin
    if (Busy or TinaCount < 2) then judy.wait;
    Busy = True;
    TinaCount = 0;
end;
```

```
procedure entry Judy-CloseJar ( );
begin
    Busy = False;
    tina.signal;
end;
```

```
Judy:
Var CookieJar-Instance : CookieJar;
While True
{ CookieJar-Instance.Judy-OpenJar ( );
  GET-COOKIE;
  CookieJar-Instance.Judy-CloseJar ( );
  EAT-COOKIE;
  DO-SOMETHING;
}
```

Now, Tina decides to be a better sister by introducing the following rule:

**TGoodness Rule:** After the first 100 cookie-jar visits by Tina, Judy is also allowed to get a cookie from the jar whenever she wants to (except, of course, when Tina is in the process of getting a cookie, Judy must wait until Tina gets her cookie).

Revise the above solution so that the TGoodness rule is satisfied.

## ONE SOLUTION:

```
type CookieJar = monitor  
int TinaCount; TotCount;  
bool Busy;  
condition tina, judy;
```

```
procedure entry Tina-OpenJar ();  
begin  
    if (Busy) then tina.wait;  
    Busy = True;  
    TinaCount++;  
    if (TotCount < 100) then TotCount++;  
end;
```

```
procedure entry Tina-CloseJar ();  
begin  
    Busy = False;  
    if (TinaCount > 1 or TotCount = 100)  
    then judy.signal;  
end;
```

```
begin TinaCount = 0; TotCount = 0;  
Busy = False end;
```

```
procedure entry Judy-OpenJar ();  
begin  
    if (Busy or (TinaCount < 2 and TotCount < 100))  
    then judy.wait;  
    Busy = True;  
    TinaCount = 0;  
end;
```

```
procedure entry Judy-CloseJar ();  
begin  
    Busy = False;  
    tina.signal;  
end;
```