CPS221 Lecture: The Critical Section Problem

last revised 9/4/12

Objectives

1. To understand the critical section problem
2. To understand various types of semaphores: binary, counting, with queue
3. To understand monitors and conditions
4. To understand how message passing can be used for synchronization

Materials:

1. Handouts of critical section problem example
2. Projectable of general structure of a program with critical section
3. Projectable of inadequate software solutions to the critical section problem, and of sequence resulting in starvation for last
4. Projectable of Peterson's algorithm
5. Projectable of bounded buffer problem solved with semaphores
6. Projectable of (incorrect) Chopstick class from Dining Philosophers using spin lock
7. Demonstration of diining philosophers problem solved with semaphores, plus code to project
8. Diagram showing basic concept of a monitor
9. Projectable of various inadequate monitor-style solutions to the bounded buffer problem (including code generated by compiler)
10. Projectable of solution to bounded buffer problem using a monitor with conditions
11. Diagram showing monitor with conditions
12. Diagram showing monitor with conditions and queue of waiting signalers
13. IncorrectThreadedRacers.java, IncorrectThreadedRacersWithDelay.java, and CorrectThreadedRacers.java to project and demonstrate
14. Demonstration of dining philosophers problem solved with synchronization, plus code to project
15. ATM Example system code to project
16. Demonstration of dining philosophers problem solved using message passing, plus code to project
17. Summary Handout

I. Introduction

A. We have been discussing the fundamental abstraction of operating systems: the notion of a PROCESS.

1. The chief task of an operating system is to manage a set of processes.

2. Most of the time, we think of the various processes on a system as being independent - i.e. each represents a separate and distinct task. In this case, the primary jobs of the operating system are to prevent the various processes from interfering with one another, and to ensure a fair allocation of the resources.

3. But, as we have already seen, it is often the case that separate processes are used to work cooperatively on a common task - either at a single location or distributed across a network Moreover, a single process may contain several threads working together on the overall task of the process. (For simplicity, we will speak of "process" throughout - with the understanding that these could be lightweight processes = threads)

B. However, concurrency gives rise to important, but challenging problems. To illustrate one of these, we will use an exercise.

1. Write three accounts and balances on the board

Account	
1234	$1000
5678	$2000
9999	$3000

2. Give out handouts for critical section demo. Half the class should get version A, half B.

3. Do part A. Are the results correct?

   ASK

4. Now do part B. Are the results correct?

   ASK

   Why is the result wrong this time when it was right last time?

   ASK

C. The problem we have just illustrated is called the critical section problem. A critical section is a region of code in which a process uses a variable (which may be an object or some other data structure) that is shared with another process (e.g. the "code" that read, modified, and wrote an account balance in the example you did.)

   Problems can arise if two processes are in critical sections accessing the same variable at the same time. (This is why the first demonstration worked correctly - the two "processes" accessed two different accounts, but the second did not because both "processes" accessed the same account)

   1. It is obvious how this problem can arise in certain cases:

      a) Cooperating processes that share memory. (Note that threads are always of this sort).

      b) Operating systems for multiprocessors or multicore computers.

2. But even on a uniprocessor that uses message passing exclusively for interprocess communication, the critical section problem can still arise with regard to the operating systems own data structures (message and process queues) if kernel code can be preempted by an interrupt from another device.

D. It may seem that the problem is rare, since it depends on timing.

1. For example, in the case of account 9999, if Group B had started its work just a few seconds later than it did, no problem would have occurred.

2. A problem that depends on coincidental timing like this is often called a race condition.

3. It is precisely this role of coincidental timing that makes the problem particularly insidious - it is difficult if not impossible to repeat a problem on demand - hence, it is important to use strategies that are provably correct in all cases.

E. To deal with critical sections, we need an approach that guarantees:

1. MUTUAL EXCLUSION: under no circumstances can two processes be in their critical sections for the same variable at the same time.

2. PROGRESS. At no time do we have a situation where a process is forced to wait forever for an event that will never occur. (This assumes, of course, that no process remains in its critical section forever, thus "hogging" the resource.) (This is also known as the NO DEADLOCK requirement.)

3. BOUNDED WAITING. No process waiting for a given resource can be forced to wait forever while other processes repeatedly access the same resource. (Again, this assumes that no process remains in its critical section forever, thus "hogging" the resource.)

(This is also called the NO STARVATION or the FAIRNESS requirement.)

II. Some Solutions to the Critical Section Problem

A. Solutions to the critical section problem are of two general types:

1. Solutions depending on special hardware facilities.

2. Solutions that are strictly software based - in the sense that the only characteristic of the hardware they rely on is that if two processes attempt to store a value in the same memory cell, then the hardware will guarantee that the final value will be the same as that written by one of the two, though nothing is guaranteed regarding the order.

B. For simplicity, we will consider the case of just two processes, each with one critical section accessing the same variable. These can be extended to handle more complex situations.

C. Software solutions: We require that each process execute special entry code before starting its critical section, and exit code upon leaving. The problem is to specify what the entry and exit code must be. That is, each process looks like this:

PROJECT

```
while(true)
{
     non-critical section;
     entry protocol;
     critical section;
     exit protocol;
     remainder (non-critical) section
}
```

D. Some inadequate solutions: (These come from different books. There are a few more than what's in your text, and I think the notation is a bit more understandable. We consider them because the mental exercise in discovering the problem in each case helps develop the sort of thinking we need to use when thinking about parallel computation.)

In each case, we assume the two processes have pids i and j. We give the code for process i; the code for process j is identical except for switching i and j.

1. (Silberschatz and Galvin fourth ed p. 167) Assume the two processes share an integer variable turn for control of access to the critical section, in addition to the critical variable(s) per se

   PROJECT

```
white(true)
{
        while (turn != i)    ;   // Spin lock
        critical section
        turn = j;
        remainder section
}
```

   Why doesn't this work?

   ASK

   Violates the progress requirement: Once one process is finished computing, the other process gets one more turn to enter its critical section, after which it can never enter its critical section again.

2. (Silberschatz and Galvin alternate ed page 86): Assume the two processes share a boolean array flag (indexed by process id) for control of access to the critical section, in addition to the critical variable(s) per se

PROJECT

```
while(true)
{
        while (flag[j])     ;    // Spin lock
        flag[i] = true;
        critical section
        flag[i] = false;
        remainder section
}
```

Why doesn't this work?

ASK

Violates the mutual exclusion requirement. Each process could see the other's flag false before either sets its flag true, allowing both to be in their critical section at the same time.

3. (Silberschatz and Galvin fourth ed p. 168) flag[] array same as in previous.

PROJECT

```
while(true)
{
        flag[i] = true;
        while (flag[j])      ;   // Spin lock
        critical section
        flag[i] = false;
        remainder section
}
```

Why doesn't this work?

ASK

Violates the progress requirement. There is a race condition - Each process sets its flag true before either examines the other's flag, so

if both processes set their flag true before either examines the other's flag, they will deadlock.

4. (Silberschatz and Peterson 1st ed) flag[] array same as in previous. This one appears correct as first, but contains a subtle error that shows just how subtle concurrent problems can be.

   PROJECT

```
while(true)
{
    flag[i] = true;
    while (flag[j])
    {
        flag[i] = false;
        while (flag[j])  ;       // Spin lock
        flag[i] = true;
    }
    critical section
    flag[i] = false;
    remainder section
}
```

   Why doesn't this work?

   ASK

   Violates the bounded waiting requirement. Can lead to starvation for both processes if they alternate in a certain pattern:

Process 0	Process 1
sets flag[0] true	
	sets flag[1] true
first while - sees flag[1] true	
	first while - sees flag[0] true
sets flag[0] false	
	sets flag[1] false
second while - sees flag[1] false	
	second while-sees flag[0] false
sets flag[0] true	
	sets flag[1] true
first while - sees flag[1] true	
	first while - sees flag[0] true
...	

PROJECT

While admittedly such the establishment of such a pattern - and its continuation over a long time - is improbable, it is not impossible. Two cores on a multicore computer, or identical CPU's sharing a common memory might well remain in such a situation if they ever enter it - a potentially insidious situation!.

E. Peterson's algorithm is a totally correct software solution to the critical section problem.

PROJECT

1. Let's demonstrate the correctness of this algorithm

   a) It guarantees mutual exclusion.

      (1) Clearly, if one of the processes is already in its critical section, the other cannot enter because the first process's

flag[] value is true and the entry protocol includes setting turn to the "wrong" value.

(2) If both processes try to enter their critical section at about the same time, only the one that sees turn having the "right" value - i.e. the one that set it to the "wrong" value first - will be able to do so.

b) It guarantees progress: For deadlock to occur, both processes would have to be stuck in their while loops. But this cannot be the case, since depending on the value of turn one of the two processes will necessarily be able to proceed. (Both turn == 0 and turn == 1 cannot be simultaneously false if the only possible values for turn are 0 and 1!)

c) Guarantees bounded waiting: Suppose that one process is starving while the other is in its critical section. (Assume without loss of generality that process 0 is the one starving - it has set flag[0] true and is stuck at the while loop.)

When process 1 exits its critical section, it sets flag[1] false. For process 0 to starve, process 1 would have to complete its remainder section and return to the top of the loop and set flag [1] true before process 0 sees it false. But, in this case, process 1 sets turn = 0 and can proceed no further since flag[0] is true. Meanwhile, process 0 will see flag == 0 and proceed.

2. Though Peterson's algorithm satisfies all our criteria, it has some limitations:

a) Complexity: three variables (turn and a two-element array) needed to control one critical section. If a program has several critical sections, the code could easily become burdensome.

b) Limited as it stands to two processes, though it can be extended to any number at the price of added complexity.

c) Relies on busy-waiting (spin locks) - hence a processor can be tied up for long times doing nothing while waiting to enter a critical section.

d) Not suited to distributed environments. For such an environment we need an algorithm that calls for each variable only to be altered by a single process, though it may be inspected by any. In this algorithm, each process must update turn when it wants to enter its critical section.

F. Hardware-based solutions.

1. The complexity of software solutions arises because we cannot, in general, guarantee that a variable will not be altered between the time that a given process looks at it and the time it itself tries to change it. This is because inspecting a value and altering a value normally require two or more machine instructions, with the possibility of an interrupt (on a uniprocessor) or an access from another processor intervening.

2. Special hardware provisions of a fairly simple sort can greatly simplify the mutual exclusion algorithms.

   a) On a uniprocessor, we can generally inhibit interrupts for a brief time. Many operating systems rely on this method totally for their own internal critical sections; and the operating system can use this method to furnish a system service to processes for their own use. (We will say more about what form this service might take shortly.) This will not work with multiprocessors, however.

b) Many processors have an instruction which tests and modifies a location in memory in a single, atomic operation, such that no other operation can intervene between the time the location is examined and the time it is modified.

For example, Intel IA32 has an XCHG operation that exchanges a register and a memory location. If one uses 1 to represent the boolean value true and 0 false, then a critical section might be protected by a single variable as follows:

entry protocol

loop:
    put 1 in EAX
    XCHG EAX and the variable protecting the critical section
    if EAX is 1, branch to loop

exit protocol:

put 0 in EAX
XCHG EAX and the variable protecting the critical section

III. **Semaphores**

A. One problem with the solutions we have looked at thus far is that they are a bit messy, and thus prone to errors in implementation.

B. A facility first proposed by Edsger Dijkstra called a SEMAPHORE builds on one of these solutions to provide a cleaner solution. In its simplest form, a semaphore is a boolean variable with two indivisible atomic operations possible on it:

```
P(s):       while (s ==  0) ;
                        s = 0;

V(s):       s = 1
```

1. The names P and V are from two Dutch words proberen (to wait) and verhogen (to increment). The text calls these wait and signal. But I will stick with P and V because wait and signal will be used in another context with a somewhat different meaning.

2. It is important to note that these operations are indivisible atomic operations, so they must either be implemented using an appropriate atomic hardware primitive or on top of a software strategy such as Peterson's algorithm. For example, P and V could be implemented with XCHG as follows

```
P(s);       loop:
                Put 1 in EAX
                XCHG EAX and s
                if EAX != 0, branch to loop

V(s)        Put 0 in EAX
            XCHG EAX and s
```

C. A generalization of the basic semaphore is called a counting semaphore, which can assume any (non-negative) integer value. The operations - which must be done indivisibly - are:

```
P(s):       while (s <= 0) ;
                    s --;

V(s):       s ++
```

1. Counting semaphores can be used for a resource of which there are a fixed number (> 1) of copies.

2. Note that the binary semaphore is a special case, with s constrained to assume only the values 0,1.

3. Counting semaphores are a bit more complex to implement. A counting semaphore could be implemented using a second, binary semaphore (called here s'):

```
P(s):       do
            {
                    P(s');
                    temp = s;
                    if s > 0 then s = s - 1;
                    V(s')
            } while temp <= 0;

V(s):       P(s');
            s ++;
            V(s')
```

D. One problem with a semaphore is that it uses busy-waiting (a spin lock). An improvement is to associate a queue with the semaphore, so that we have:

```
P(s):       s --;
                    if s < 0 then block this process in s's queue
```

V(s): s ++;
if s <= 0 then unblock one queued process

1. When this strategy is used with a semaphore whose initial value is 1, the result is binary semaphore behavior.

2. As a consequence of this definition, the value of s has the following meaning

   a) If s > 0, it is the number of processes that can do P() without blocking.

   b) If s < 0, it is the number of processes that are blocked on the semaphore.

3. In the literature, it is common to find that no assumptions are made about the behavior of the queue - i.e. it is not necessarily a true, FIFO queue. In practice, though, it is sometimes desirable to require fair (FIFO) behavior for the queue - in which case the description of the semaphore will say this explicitly.

   (A semaphore with a FIFO queue associated with it is the variant we will use in our examples.)

4. How do we implement a semaphore with a queue? We treat it as a critical section in its own right, with its mutual exclusion guaranteed by a lower level method such as a binary semaphore or one of the software schemes. (Here the busy waiting is very short and can be tolerated.)

E. Some examples of using semaphores to solve classical problems

1. The bounded buffer problem

   PROJECT code from book

a) mutex is a binary semaphore, while empty and full are counting semaphores representing the number of empty and full slots in the buffer, respectively. Therefore, empty is initialized to the buffer size, while full is initialized to 0.

b) Why is mutex needed along with empty and full?

   ASK

   mutex is needed for mutual exclusion. Without it, if the buffer is partially full it might happen that the producer and consumer both try to update the buffer at the same time

c) Go over logic of code

2. The Dining Philosophers.

   a) In lab we solved this problem using a spin lock - which (as was pointed out then) is a poor solution due to wasting CPU cycles and is not actually a correct solution due to a potential timing problem.

      PROJECT incorrect Chopstick class with spin lock

      Can anyone see the problem?

      ASK

      Though unlikely, the following sequence of events *could* occur:

      One thread sees `owner == null` - leaves its while loop
      A second thread sees `owner == null` - also leaves its while loop
      First thread sets `owner = who` (itself)
      Second thread sets `owner = who` (itself)

Now *both* threads think they are the owner of the resource - the problem we were trying to avoid.

Note that a concurrent program is only considered correct if there is no timing-dependent sequence of operations that could make it fail!

b) A totally correct solution can be created using semaphores.

DEMO

c) Go over code in class Philosopher. (Note: we have totally replaced the Chopstick class used in lab by a semaphore, with the `pickup()` and `putdown()` operations on the chopstick being changed to semaphore operations `P()` and `V()`),

F. Problems with the semaphore solution to the critical-section problem:

ASK

1. The burden is on the programmer to use the semaphore. There is no way to stop a programmer from updating a shared variable without first doing the necessary P(), short of manually inspecting all code. Thus, mutual exclusion can be lost through carelessness or laziness

2. There is a danger of accidentally doing P() on the wrong semaphore, thus gain losing mutual exclusion.

IV. **Message-Passing**

A. When we talked about processes and threads, we noted that there are two basic approaches to interprocess communication. What were they?

ASK

Shared memory
message passing.

B. The critical section problem is peculiar to the shared memory model, but also arises in the operating system code used to support message passing (since the message queues typically are accessed by multiple processes on the same computer. The critical section problem is typically avoided for application code in the message passing model by: having each shared variable be "owned" by one process. Any other process wishing to change it must do so by sending a message to the owner process.

C. However, the need for synchronization between processes still arises. This is typically handled by means of exchanging messages according to some protocol.

Example: A message passing solution to the Dining Philosopher's problem

1. DEMO

2. PROJECT Code for Philosopher

3. PROJECT Code for coordinatorLoop() in Dining Room

D. The basic idea is this: the operating system makes the following primitive operations available to processes:

1. SEND destination-specifier message

2. RECEIVE [source-specifier] message

   (Explicit specification of the source of a received message is possible on some systems but not others)

   Message is usually a sequence of bytes of arbitrary length whose interpretation is determined by the cooperating processes (i.e. the operating system simply passes the bytes from one process to the other.)

E. The designer of a message passing system must consider quite a number of basic questions, including the following:

   1. How does a sender specify a destination?

      a) Some systems set up a communication CHANNEL between processes, so that the sender specifies the destination by channel. (This is analogous to the idea of opening a file by name and then doing all further IO operations by specifying the channel that was named in he open call.)

      b) Other systems require the sender to specify the name or process id of the intended recipient.

   2. If communication channels are used, are they tied to two specific processes?

      a) Most systems allow several different processes to send messages on the same channel.

      b) Some systems allow several different processes to receive from the same channel. (In this case, the sender may not necessarily

know which process will actually get the message. This is fine if all possible recipients provide the same basic service.)

c) If several processes can receive from the same channel, another question that must be answered is: does each message go to all processes connected to that channel or only to the first receiver? (The answer is usually the latter.)

3. Can several messages be queued in one channel or for one process? If so, is there any guarantee as to the ORDER in which they will be delivered to subsequent receiver operations (FIFO)?

4. Is the sender of a message forced to wait until the receiver has responded before proceeding?

   a) Some systems require this;

   b) Others make it an option - in which case a third primitive may be supported:

      SEND_AWAIT_REPLY destination-specifier message

5. If a process tries to do a receive when no messages are available, does it wait until a message arrives?

   a) Many systems allow a receiver to specify that it wants to continue execution of no messages are waiting. In this case, it will have to try the receive operation again later.

   b) Many systems allow a receiver to specify a timeout period, such that it will wait for a message for a certain period of time, after which it will give up.

c) Some systems provide a facility whereby an incoming message can interrupt the execution of the receiver to notify it that a message is waiting.

6. If a receiver can specify a channel on which it wants to receive a message, can it specify some sort of LIST of channels, such that if a message is pending on any of them it will get it?

7. If a receiver can specify several different possible channels in a receive operation, is there any guarantee as to which channel it will receive from if messages are pending on several of them?

F. Example: Unix interprocess communication.

1. Interprocess communication is done using a called a SOCKET. Sockets can be used for interprocess communication on the same system and also to send messages over a network to processes on another system.

   (The pipeline facility we discussed earlier is a much less general precursor to sockets. On modern systems, pipelines are actually implemented using sockets to link the processes)

2. The socket facility supports a variety of PROTOCOLS regarding message format, etc. We will describe only one - in which messages are arbitrary-sized packets of bytes.

3. The following system services are used for accessing sockets:

   a) socketpair creates a pair of connected sockets.

      (1) What is sent on one end comes out the other end and vice versa. (Therefore, a process can write to its end of a pair of connected sockets and then read from the same socket without getting its own message back.)

(2) Normally cooperating processes are set up by a series of fork operations from a common parent. In this case, the parent will normally create the socket pairs as it creates the processes, and the children will inherit them from the parent.

b) Alternately, individual sockets may be created by the socket service and connected via the connect service. In this case, the bind and getsockname services may be used to give a socket a name and to find out the name of a socket of interest. Depending on the protocol in use, multiple connections to the same socket are possible.

4. In the simplest approach to using sockets (using the STREAM protocol), data is sent and received as a stream of bytes. This means that no internal record is kept in the socket as to where one message ends and another begins; thus, the receiver must know how long an incoming message is before it can receive it correctly. (This is normally handled by agreement between the sender and receiver.)

5. The send and recv services transmit data over sockets. Send does not wait until the message is received (but may wait if there is no room for additional data in the stream); recv may or may not wait if no data is available, depending on the setting of options on the socket when it is created.

6. The select service can be used to find out which sockets from a list specified by the caller currently have messages pending (if any). The recv service is then used to actually get the individual messages one at a time in whatever order the receiver wishes to.

V. **Monitors**

A. While semaphores do provide a shared memory solution to the critical section problem, they are not an ideal solution because of the possibility of programmer error, such as forgetting to use P() and V(), or using them incorrectly.

B. An improvement on semaphores is the MONITOR.

1. At about the same time that pivotal work was being done on concurrency primitives, another area of research was developing the concept of data abstraction. The key idea is that we define an abstract data type in terms of a set of visible procedures and a set of hidden implementation code. The notion of class in OO is one realization of this concept.

2. The monitor concept is a fusion of ideas from abstract data types and a notion called critical regions which we will not separately discuss.. A monitor provides the rest of the world with a limited set of mechanisms for accessing some protected data.

3. A monitor resembles an OO class, but is implemented in such a way as to allow only one process to be executing any of its methods on a given object at any time.

   a) Conceptually, the compiler for a programming language that includes monitors will create a single "mutex" semaphore (with FIFO queue) for the monitor, and will insert P and V operations at the start and end of each method of the monitor.

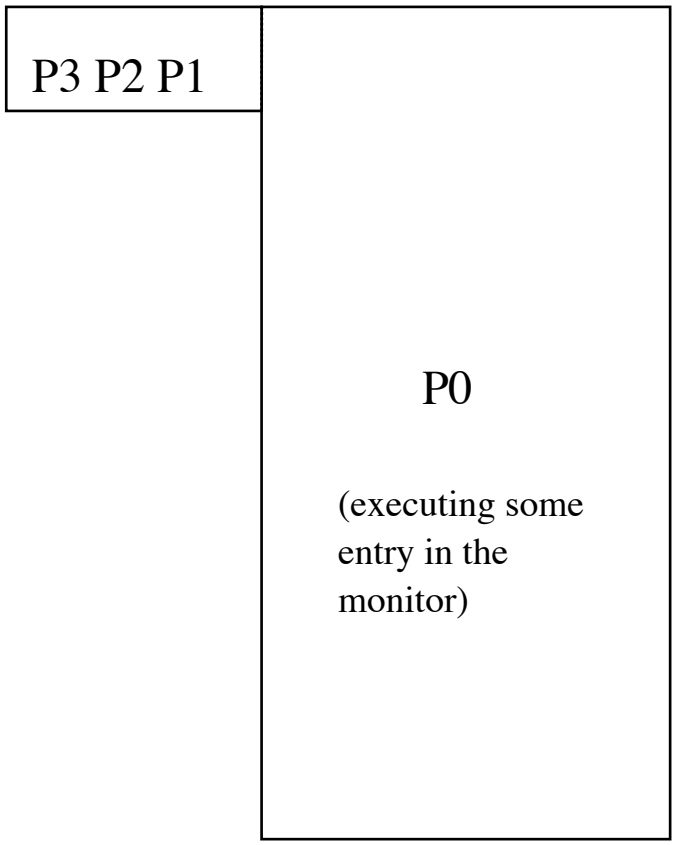
   b) While research programming languages that fully implement monitors have been created, no regular programming language does - though languages like Java have facilities based on them.

c) We will first discuss monitors conceptually, and will then discuss their realization in Java.

4. A monitor may be pictured like this:

PROJECT

Queue of processes
waiting to execute
some entry



when the monitor entry code P0 is executing finishes, P1 will be allowed to execute its entry (with may not be the same as the one P0 was executing.)

C. Example: the bounded buffer problem solved using a monitor. (Note: this solution is incomplete - but demonstrates the need for a feature we will add shortly.)

1. We would have a monitor named buffer with two entries:

   insert - put a character in the buffer
   remove - remove and return a character from the buffer

   The code we would write would something like this in a programming language that includes monitors:

   PROJECT

```
monitor buffer
{
        -- some suitable collection of characters

        void insert(char c)
        {
             put c in the buffer;
        }

        char remove()
        {
             remove a character from the buffer and
                  return it;
        }
}
```

   The monitor code generated by the compiler will guarantee that at most one of these entries is active at any one time

2. The compiler would add a semaphore and P and V operations so that the code actually implemented would look like this. (Code added by the compiler in boldface)

   PROJECT

```
class buffer
{
        Semaphore mutex = new Semaphore();
        -- some suitable collection of characters

        void insert(char c)
        {
             P(mutex);
             put c in the buffer;
             V(mutex);
        }

        char remove()
        {
             P(mutex);
             char c;
             remove a character from the buffer and
                  put in c;
             V(mutex);
             return c;
        }
}
```

D. As we noted earlier, something is missing from this solution. What?

ASK

1. There is no provision for ensuring that the producer doesn't try to insert a character into a full buffer, or that the consumer doesn't try to remove a character from an empty buffer. Obviously, this is a problem we need to solve!

2. Here are a couple of inadequate ways to attempt to solve this problem.

   a) Add a check on the buffer status with a spin lock to each entry - e.g. insert() could be changed to:

```
void insert(char c)
{
     while (buffer is full) ; // spin lock
     put c in the buffer;
}
```

(and remove() could be changed similarly).

(1) But this will not work as we want. Why?

ASK

(2) The code generated by the compiler for this method would be:

PROJECT

```
void insert(char c)
{
        P(mutex);
        while (buffer is full) ; // spin lock
        put c in the buffer;
        V(mutex);
}
```

which means the spin look lies inside the critical section protected by the semaphore, which means that if a process entered this loop it would retain mutual exclusion and thus the consumer could never get into the monitor to create a free space! As a result, we would have deadlock!

b) Add a method to the monitor to check whether the buffer is full (and likewise a method to check whether it is totally empty.) The producer code that uses insert() would be modified to look like this:

```
while (buffer.isFull()) ; // spin lock
buffer.insert(character);
```

(and similarly for the consumer code that calls remove()).

(1) While this would put the spin lock outside of any monitor entry, eliminating the above problem, and would work for this specific example, it would create a new problem if we had more than one producer process What?

ASK

(2) If there were more than one producer process (or the like in some other problem), then there could be one space free in the buffer which two producers see as available - but the second one to actually call insert() would end up trying to put a character in a full buffer!

(3) In other words, there cannot be a "break" in mutual exclusion between the check and the actual insert.

E. To cope with problems like this, the notion of monitors includes another type of variable called a CONDITION, with operations wait and signal.

1. The condition is superficially like the semaphore, but has three important differences:

   a) A process executing a wait is ALWAYS blocked.

   b) A signal executed when the queue for the condition is empty has no effect - it is not remembered; it is simply ignored.

   c) Also, the queue of waiters on a given signal is assumed to be FIFO.

2. With the addition of condition variables, we can give a complete solution to the the bounded buffers problem using a monitor:

   PROJECT

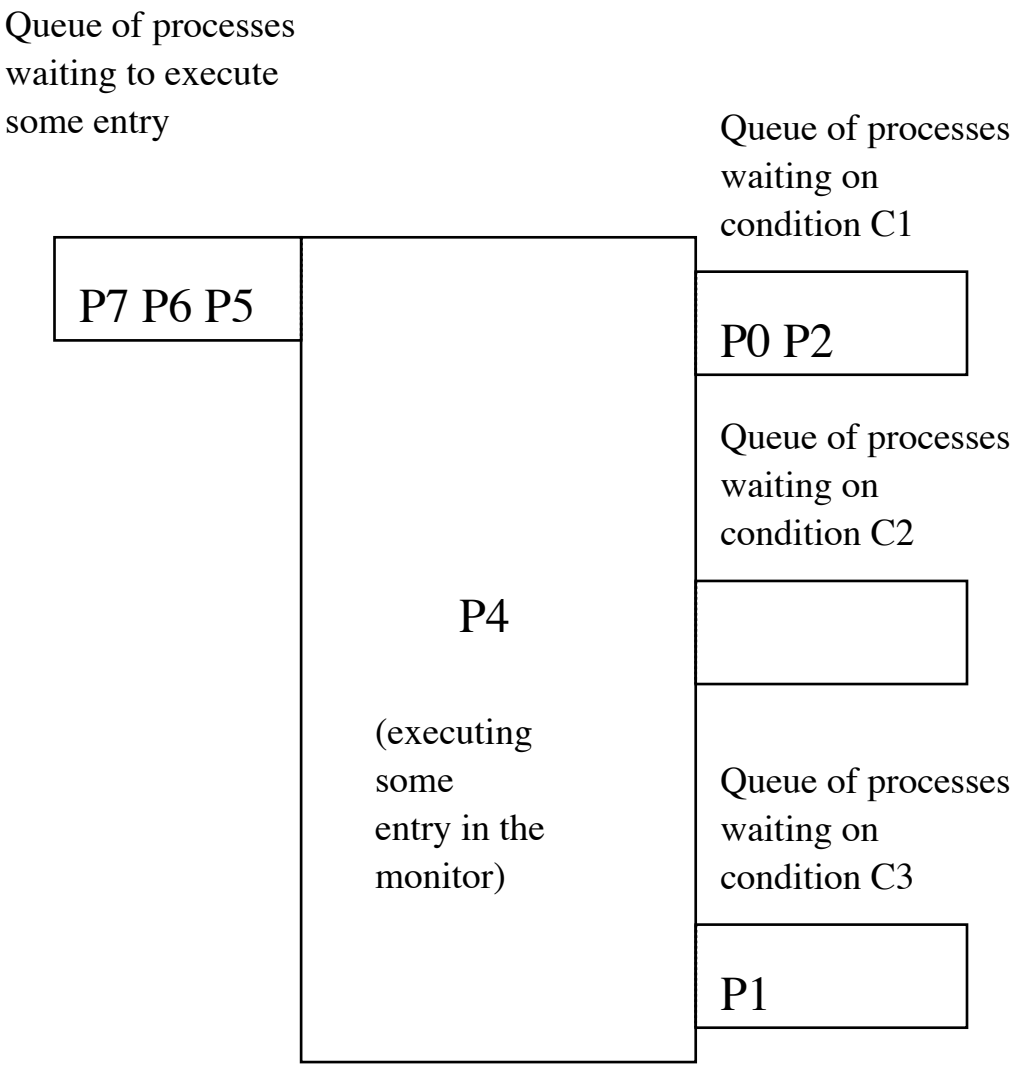
```
monitor buffer
{
        -- some suitable collection of characters
        Condition notFull, notEmpty;

        void insert(char c)
        {
             if (the buffer is full)
                  wait(notFull);
             put c in the buffer;
             signal(notEmpty);
        }

        char remove()
        {
             if (the buffer is empty)
                  wait(notEmpty);
             remove a character from the buffer and
return it;
             signal(notFull);
        }
}
```

3. In the simplest variant of this, we impose the restriction that a signal operation - if it appears in a given monitor entry - must appear at the very end. When a signal is executed, the calling process leaves the monitor, and the first process on the queue awaiting that condition is admitted, taking its place. We can picture such a monitor as follows:

   PROJECT



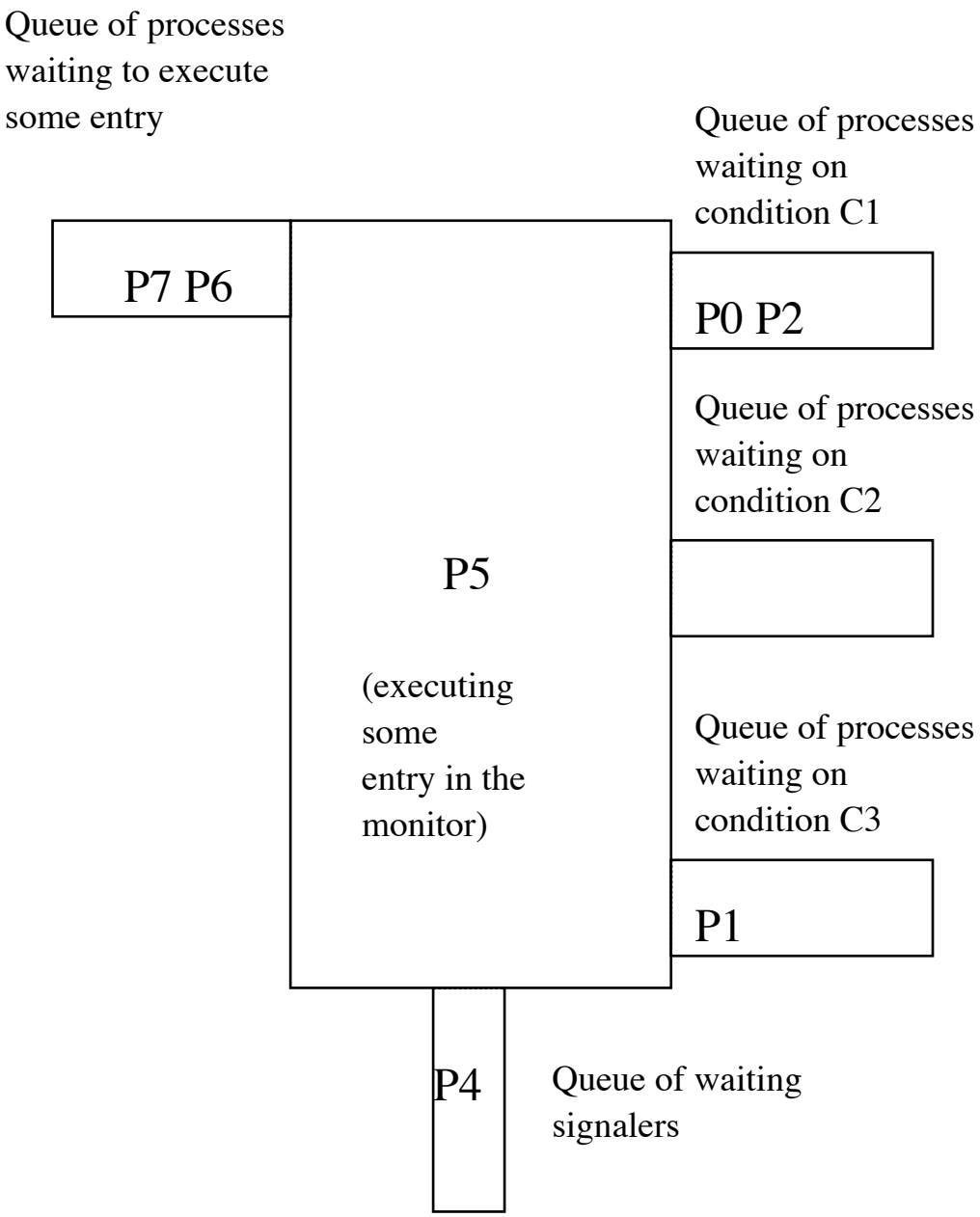
a) If P4 signals C1 at the end of its entry, P0 will resume executing its entry.

b) If P4 signals C3, P1 will resume its entry.

c) f P4 signals C2, or finishes its entry without signaling any condition, P5 will be allowed to execute its entry.

4. A variant of this removes the restriction that a monitor entry can only signal as its last act. (This would also allow a given entry to generate more than one signal.) This raises a problem - though - when a signal awakes a process, and the signaler wishes to remain in the monitor, one must yield to the other. One way to handle this is as follows: if an entry contains a signal other than as its last statement, the calling process is suspended while the awakened process completes its work. The calling process has priority to re-enter the monitor over any processes awaiting at the main gate. This can be pictured as follows:

   PROJECT



a) If P5 signals C1 at the end of its entry, P0 will resume executing its entry.

b) If P5 signals C3, P1 will resume its entry.

c) f P5 signals C2, or finishes its entry without signaling any condition, P4 will be allowed to resume executing its entry.

F. Monitors in Java

1. We have previously discussed support for threads built into Java. We now turn to Java's mechanisms for handling the critical section problem. Java's approach is based on the monitor concept, but is not a strict implementation of it. (In fact, to my knowledge, no regularly-used programming language implements monitors as we have just discussed.)

2. In Java, every variable has a monitor associated with it, though the monitor is not used for most objects. (In fact, the monitor may not even be created until the first time it is used.)

3. The synchronized statement allows a block of code to be executed under mutual exclusion using the object's monitor -

   a) e.g. in the following code:

```
bar();
synchronized(foo)
{
     baz();
     frobbish();
}
foobar();
```

   while he methods baz() and frobbish() are being executed, no other code synchronized on the object foo may be executed; but this is not true while either bar(); or foobar() is executed.

   b) If a thread attempts to enter a synchronized block on some object while another thread is in a synchronized block on the same object, it is forced to wait until the previous thread has finished its work.

c) It is also possible to declare a method synchronized. This is equivalent to synchronizing on the "this" object of the method.

Example:

```
synchronized void something()
{
     ...
}
```

is equivalent to

```
void something()
{
     synchronized(this)
     {
          ...
     }
}
```

d) Note that synchronization locks the the monitor of a specific object - not a method.

Example: consider the following

```
class SomeClass
{
     synchronized void m1()
          ...
     synchronized void m2()
          ...
}

SomeClass o1, o2;
```

Now suppose we have threads executing code as follows:

Thread 1: `o1.m1();`
Thread 2: `o2.m1();`
Thread 3: `o1.m2();`

Threads 1 and 2 can execute their code simultaneously, because they are using different objects and hence different monitors. But Threads 1 and 3 cannot execute their code simultaneously because though they are using synchronized methods of the same object.

4. To see how to use synchronization in Java, we will extend the "Racer" example we did earlier.

   Consider the racer program again. As it stands, each racer keeps running until it completes, so we have no way of knowing who won except by careful observation.

   a) Suppose, instead, we add a `StringBuffer` variable to the main class, that allows the winning thread to report its name. (Note that all four racers *share* the *same* `StringBuffer`.)

      (1) We set the initial contents of the `StringBuffer` to empty.

      (2) When a thread finishes, it checks to see if the `StringBuffer` is empty. If it is, it writes its name into the `StringBuffer`. (We have to check first, else threads that finish later will overwrite the name of the winner). At the end of the race, the main program writes the results. The code to do this can be added to the end of the `run()` method of the racers.

      SHOW CODE - `run()` method in IncorrectThreadedRacers.java

      DEMO

b) Is this code correct?

ASK

Surprisingly, the answer is no! Although it will work correctly most of the time, it can sometimes produce the wrong result. Consider the following scenario: suppose Red finishes with Green close behind it. Suppose, further, that due to the way the threads are scheduled, the following series of events occurs:

(1) Red finishes, and checks to see if the `StringBuffer` is empty - it is.

(2) Green finishes, and checks to see if the `StringBuffer` is empty - it still is.

(3) Red writes its name into the `StringBuffer`

(4) Green writes its name into the `StringBuffer`.

(5) Although Red won, Green is reported as the winner!

(6) It may be argued that this scenario depends on the race being very close, and even then is improbable. Try telling that to runners in the Olympics! The fact that a scenario like this is rare does not mean its impossible, and the insidious thing is that finding such an error during testing, or making it repeat itself during debugging, is virtually impossible. Thus, the only way to produce correct concurrent software is to make sure such a scenario *cannot* occur.

c) To see that this is really a problem, we will run a version of the program that has been modified to insert some extra delay into

the finishing code, between the time that the thread checks the contents of the `StringBuffer` (and sees that it's empty) and the time that the thread writes new content into it.

DEMO: IncorrectThreadedRacersWithDelay

SHOW CODE at end of `run()` method - note that the logic is the same, but that delay loops and println's have been added.

d) Now, we will look at a version of this program that uses synchronization to achieve correct results.

SHOW CODE - CorrectThreadedRacers.java - end of run() method.

DEMO

5. A weakness in the Java solution to the critical section problem is that while one thread has locked an object, other threads can still access it through code that is <u>not</u> specified as `synchronized`.

a) Java is defined this way because locking an object involves a fair amount of overhead, so we don't want to do it unless we have too.

b) However, this leaves open the possibility that a programmer might forget to specify that a given section of code is synchronized when it should be, negating the protection afforded by declaring some other section of code for the same object to be synchronized.

(Sort of like the possibility that one roommate might lock the door of the room and the other roommate might forget to lock it, leaving the first roommates' stuff vulnerable.)

6. What about support for something like conditions in Java? For each monitor, Java supports what amounts to a single condition through the methods `wait()`, `notify()`, and `notifyAll()` that are defined in class `Object`, and therefore available for all objects.

   a) A thread that holds a lock on some object can execute the `wait()` method of that object. When this occurs:

      (1) The thread's lock on the object is released, so other threads can access it.

      (2) The thread that executed the `wait()` is rendered unable to proceed - it is said to be *blocked* on that particular object.

   b) Some other thread (which must now hold the lock on this object) may subsequently execute the locked object's `notify()` method.

      (1) When this occurs, one thread that was blocked on the object is unblocked. (If several threads are blocked on the same object, there is no guarantee as to which is unblocked.)

      (2) The thread that was unblocked may proceed after re–obtaining the lock on the object.

   c) It is also possible to use the `notifyAll()` method of an object to unblock *all* threads that are blocked on that object - though they will, of course, have to proceed one at a time since a thread that was blocked must re-obtain the lock on the object before it can proceed.

   d) The `wait()` method can optionally specify a timeout value. If it is not notified within this time period, the thread is unblocked anyway.

e) To illustrate how this mechanism can be used, consider a Java solution to the Dining Philosophers problem using synchronization with wait and notify. (This is actually the typical way to solve this problem in Java, rather than using spin locks or semaphores as we have demonstrated previously, or message passing as we will discuss shortly)

(1) DEMO

(2) PROJECT Chopstick class from Java solution

7. Another place where `wait()` and `notify()` is useful is in cases where we want to avoid having the awt event thread do extensive computation. In this case, we use a separate thread to do the computation, which waits until the awt thread notifies it that an appropriate event has occurred.

   There are several examples of this in the ATM Example system.

   a) *EXAMPLE: SHOW* State Diagram for class ATM, then

   SHOW CODE for class ATM

   (1) Note that ATM implements `Runnable`, and has a `run()` method. When the simulation starts up, a `Thread` is created to execute this (i.e. there is a special thread for actually running the simulation.)

   (2) The `run()` method uses `wait()`/`notify()` in two places:

   (a) When in the OFF_STATE, the thread waits. It will be notified by a call of `switchOn()` by the awt thread.

(b) When in the IDLE_STATE, it waits. It will be notified either by a call to `cardInserted()` or by a call to `switchOff()` from the awt thread

b) Threads with wait/notify are also used for the simulation of the Keyboard and the EnvelopeAcceptor.

(1) In the former case, the main thread waits until the user clicks a button simulating one of the keys on the keyboard.

(2) In the latter case, the main thread waits until the user clicks the button to insert the envelope - or until a timeout occurs.

SHOW CODE for `acceptEnvelope()` method in class `SimEnvelopeAcceptor`.

8. Java 1.5 added the package java.util.concurrent and its subpackages, which provide more complete concurrency support (including both full-blown monitors and semaphores, among other things) for cases where this is needed.

VI. **Summary Handout**