CPS323 Lecture: Concurrency

Materials:

1. Coroutine Projectables
2. Handout for Ada Concurrency + Executable form (simple_tasking.adb, producer_and_consumer.adb)

I. Introduction

A. Most of our attention throughout the CS curricum has been focussed on _sequential_ programming, in which a program does exactly one thing at a time, following a predictable order of steps. In particular, we expect a given program will always follow the exact same sequence of steps when run on a given set of data.

B. An alternative is to think in terms of _concurrent_ programming, in which a "program" (at least in principle) does multiple things at the same time, following an order of steps that is not predictable and may differ between runs even if the same program is run on exactly the same data.

C. Concurrency has long been an interest in CS, but has become of particular interest as we move into a situation in which raw hardware speeds have basically plateaued and multicore processors are coming to dominate the hardware scene. There are at least three reasons why this topic is of great interest.

   1. Achieving maximum performance using technologies such as multicore processors.

   2. Systems that inherently require multiple processors working together cooperatively on a single task - e.g. distributed or embedded systems.

   3. Some problems are more naturally addressed by thinking in terms of multiple concurrrent components, rather than a single monolithic program.

D. There are two broad approaches that might be taken to concurrency:

   1. Make use of system libraries, without directly supporting parallelism in the language. (The approach taken by most languages - e.g. the use of the pthreads or PVM or MPI with a language like C++).

   2. Incorporate facilities for specifying parallel computation in the language itself (e.g Java, C#, Ada).

E. This leads us to ask what features are or might be present in a programming language to support concurrency? Of course, there are a multitude of possible answers to this question.

   1. We have already looked at once sort of answer: multi-threading as found in languages like Java.

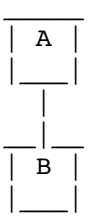
With concurrent threads, in principle more than one computation may be happening at the same time, but in practice this might not actually be the case. Even so, since switches between threads can occur at any time, careful attention must be paid to synchronizing threads so they don't interfere with one another.

2. We will look at two more representative models today. One is a fairly old idea; the other a facility found in Ada and other languages used for embedded systems.

II. Coroutines

A. Ordinarily, when we think of segmenting a program into procedures, we think HIERARCHICALLY.

1. Example: If A calls B, we have



a. A's execution pauses to let B begin.

b. B's local data is created from scratch.

c. B runs to completion.

d. B's local data is destroyed.

e. A resumes where it left off.

2. If A calls B again, the whole process is repeated.

a. The first execution of B has no effect on the second (except possibly through global data).

b. B always starts execution from the very beginning, and runs through to the end. There is no way for B to "pause in mid-stream" and return control to A, picking up later where it left off.

B. This hierarchical pattern is just what we want for most applications. However, there are some problems for which it doesn't work well, because it is just not possible to establish a clear-cut hierarchy. A good example of this sort of problem is the classic problem of the producer and the consumer.

1. In one version of this problem, a data copying program is to be written that reads data from punched cards and writes it to a disk file, subject to the following formatting requirements. (Of course, the use of punched cards for input is from the early days of computing; but the problem issues remain valid and this example has become traditional.)

a. Each punched card contains exactly 80 characters.

b. A CR/LF sequence is to be appended to the data from each card before writing it to the disk.

c. Data is to be written to disk in blocks of 512 characters. The last block may need to be padded with null characters if it is not completely filled by data from the cards.

2. We can formulate this problem in terms of two subtasks, called the producer and the consumer, where the consumer "produces" characters by reading them from cards or manufacturing them, and the consumer "consumes" characters by writing them to the disk.

   a. The producer executes the following algorithm: PROJECT

```
do
    read a card into card_buffer[80];
    for (int = 0; i < 80; i ++)
        send card_buffer[i] to the consumer;
    send CR to the consumer;
    send LF to the consumer
while (! out of cards);
```

   b. The consumer executes the following algorithm:

```
do
    for (int j = 0; j< 512; j ++)
        if there is another character available then
            accept a character from producer into disk_buffer[j]
        else
            disk_buffer[j] = NULL;
    write disk_buffer[] to disk
while (more characters are available)
```

3. While each of the above is nice and modular, we run into serious problems when we try to put these two algorithms together in a standard hierarchical way.

   a. We could make producer call consumer each time it produces a character.

      i. In this case, producer would call consumer from 3 different places (the three lines that say send --- to consumer.)

      ii. But consumer would need to be entered in the middle of a loop!

      iii. Further, consumer would have to get control one last time after producer finishes to pad out and write the last block.

      iv. Consumer's local variables j and disk_buffer would have to be preserved between calls - either by using globals or (in a language like C) by declaring them static.

   b. Alternately, we could make consumer call producer each time it needs a character.

      i. Now consumer calls producer from just one point, in mid-loop.

      ii. But producer needs to be capable of being entered at any of three places, depending on where it is in the processing of a card.

iii. In this case, producer's local variables i and card_buffer must be preserved between calls.

4. The following is one way to solve the problem hierarchically, with consumer "on top" calling producer. It uses an additional "state" variable to keep track of the three possible ways producer can generate characters:

PROJECT

```
bool no_more_characters;

char producer()
{
    static enum { READ_NEXT, CARD_NEXT, CR_NEXT, LF_NEXT }
        pState = READ_NEXT;
    static int i;
    static char card_buffer[80];

    switch(pState)
    {
        case READ_NEXT:
        {
            read a card into card_buffer[];
            char c = card_buffer[0];
            i = 1;
            pState = CARD_NEXT;
            return c;
        }
        case CARD_NEXT:
        {
            char c = card_buffer[i];
            i ++;
            if (i >= 80)
                pState = CR_NEXT;
        }
       case CR_NEXT:

            pState = LF_NEXT;
            return CR;

       case LF_NEXT:

           if there are no more cards available then
                no_more_characters = true;
           pState = READ_NEXT;
           return LF;
    }
}
```

```
void consumer()
{
    char disk_buffer[512];

    do
    {
        for (int j = 0; j< 512; j ++)
            if (no_more_characters)
                disk_buffer[j] = NULL;
            else
                disk_buffer[j] = producer();
            write disk_buffer[] to disk
    } while (! no_more_characters);
}

void main(int argc, char ** argv)
{
    no_more_characters = false;
    consumer();
}
```

5. This solution is far from elegant.

   a. We have transformed control flow information into data in the form of the variable pState plus a switch statement.

   b. The producer makes use of several static variables to hold state information. In non-C languages, these would typically have to be global.

C. To deal with cases like this (which do occur quite often in practice), some languages offer a special construct known as a COROUTINE.

1. In contrast to a procedure which is always executed from top to bottom, a coroutine can:

   a. Execute a resume statement at any point. This returns control to another coroutine, WITHOUT DESTROYING THE COROUTINE'S LOCAL STATE OR VARIABLE INFORMATION.

   b. Be resumed by another coroutine, picking up exactly where it left off.

2. The new feature being introduced is some form of RESUME statement, which yields control without destroying local state information the way return does.

3. Using coroutines, our problem could be coded as follows, using two coroutines and two global variables for communication between them. (Note: the code here doesn't correspond to any existing language. However, it could be translated into Modula-2 with little difficulty.)

   PROJECT

```
var
    C: char;
    no_more_characters: boolean;

coroutine producer;

    var
        i: integer;
        card_buffer: array[1..80] of char;

    begin
        no_more_characters := false;
        resume(main);
        repeat
            read a card into card_buffer[1..80];
            for i := 1 to 80 do
              begin
                c := card_buffer[i];
                resume(consumer)
              end;
            c := CR;
            resume(consumer);
            c := LF;
            resume(consumer);
        until there are no more cards available;
        no_more_characters := true
    end;

coroutine consumer;

    var
        j: integer;
        disk_buffer: array[1..512] of char;

    begin

        repeat
            for j := 1 to 512 do
                if no_more_characters then
                    disk_buffer[j] := NULL
                else
                  begin
                    resume(producer);
                    disk_buffer[j] := c
                  end;
            write disk_buffer[1..512] to disk
        until no_more_characters

    end;

begin
    start(producer);
    start(consumer)
end.
```

D. Coroutine facilities are relatively rare, though they have been present in an extended version of Algol, Simula, and Modula-2.

   1. One reason why coroutine facilities are relatively rare is that they are complicated to implement.

   2. The resume instruction itself is fairly easy to implement, as follows:

      a. First, the current coroutine pushes its own next instruction address onto its own stack.

      b. Then, a switch is made to the stack of the new coroutine.

      c. At this point, an address is popped from the stack of the new coroutine, and control is transferred to it.

      d. That is, the resume combines features of BOTH routine call AND routine return.

   3. The real difficulty lies in storage management. Ordinary procedures obey a LIFO discipline; thus, local storage for them can be managed by a single run time stack. But this won't work for coroutines; instead, each coroutine needs ITS OWN stack.

      a. This poses a problem with regard to stack growth, since each stack has neighbors that limit how big it can get.

      b. Some coroutine implementations require that each coroutine pre-declare the maximum stack space it will need, and this amount of space is set aside for its use. This can be dangerous: an underestimate can lead to one coroutine corrupting another's stack.

      c. Other implementations let the current coroutine use the regular system stack, but copy its stack to a holding area when it resumes another coroutine. This can make the resume operation very time-consuming.

E. An alternative way of solving this sort of problem is the use of threads or full concurrency.

   1. A fundamental difference between threads and coroutines is that with coroutines

      a. Exactly one coroutine runs at any time. (This is also true of threads on a uniprocessor, but on a multi-core processor it is conceivable that two threads might literally run at the same time.)

         (Thus, coroutines are of interest for modularity reasons, but do not yield performance improvements.)

      b. Coroutines explicitly yield control to each other, whereas threads may switch at any time. As a result, the kinds of synchronization issues that happen with threads don't arise as much with coroutines, because control transfer is predictable.

   2. Actually, a uniprocessor implementation of threads may actually implement them as coroutines, but with some mechanism for forcing transfer of control rather than doing this explicitly.

F. A variant of the coroutine known as a GENERATOR is also a key feature of Icon, and something like it can be done in Prolog.

Example: (Demo)

```
natural(1).
natural(N) :- Natural(X), N is X + 1.
```

natural(N) - keep entering ; to generate successive values

C. The Ada approach is quite different from that used in, say, Java or C#.

1. The Java approach is designed for shared-memory parallelism using threads - that is to say, the threads making up a concurrent program cooperate with one another by modifying memory that they share in common.

   An implication of this is that threads necessarily run on the same system.

2. The Ada approach is built around message passing Ada tasks cooperate with one another by sending messages back and forth, rather than through shared memory.

   The tasks could run on the same machine using threads, but they could also run on different machines, communicating via some interconnection structure or network.

   This makes it possible to use the Ada tasking model in embedded systems which often have multiple processors responsible for controlling different parts of the system.

3. Actually, Ada 95 also provides support for shared variables, but that's an advanced feature we won't get into.

4. Note that both of these approaches are coarse-grained; neither approach supports instruction level parallelism - which in any case is not available on general-purpose computers (only research systems) at this point anyway. Thus, the facilities we will look at today support distributed computation and use of parallelism as a modularity strategy - but NOT instruction-level parallelism.

III. Concurrency in Ada

A. The Ada approach is quite different from that used in, say, Java or C#.

1. The Java approach is designed for shared-memory parallelism using threads - that is to say, the threads making up a concurrent program cooperate with one another by modifying memory that they share in common.

   An implication of this is that threads necessarily run on the same system.

2. The Ada approach is built around message passing Ada tasks cooperate with one another by sending messages back and forth, rather than through shared memory.

   The tasks could run on the same machine using threads, but they could also run on different machines, communicating via some interconnection structure or network.

   This makes it possible to use the Ada tasking model in embedded systems which often have multiple processors responsible for controlling different parts of the system.

3. Actually, Ada 95 also provides support for shared variables, but that's an advanced feature we won't get into.

4. Note that both of these approaches are coarse-grained; neither approach supports instruction level parallelism - which in any case is not available on general-purpose computers (only research systems) at this point anyway. Thus, the facilities we will look at today support distributed computation and use of parallelism as a modularity strategy - but NOT instruction-level parallelism.

B. Ada has a language construct known as a TASK. (This notion was previously introduced in PL/1, but in a less-developed form.)

1. A non-concurrent Ada program contains a single main procedure that is started when the program is started; and when the procedure terminates the program terminates.

2. A concurrent Ada program can include any number of tasks - declared by task ... is instead of procedure ... is. When the program is started, each task is started, and all tasks run in parallel with one another and with the main program. Of course, some tasks may terminate while others are still running. When all tasks terminate (including the main program), the overall program terminates.

   HANDOUT; simple_tasking.adb

   DEMO

3. The Ada syntax for task declaration is actually quite complex - one can create data objects of type task, and can thus have arrays of tasks etc. We will not go into this now. The example we looked at is about as simple as it gets!

C. Ada tasks communicate via MESSAGE PASSING.

1. Example (not using Ada syntax yet):

```
task body Task1 is
    ... send somemessage to Task2
end Task1;

task body Task2 is
     ... accept somemessage
end Task2;
```

2. This example, though very simple, exemplifies a key point: both tasks need to do something for communication to take place - one must send a message, and the other must accept it.

3. The Ada mechanism for handling this called a RENDEZVOUS.

D. The Ada Rendezvous

1. In our introduction to Ada tasking, we saw that tasks (like packages) have a two part declaration: a task specification and a task body.

   a. In the first example we looked at, the task specification was empty (though still required by the syntax.)

   b. In general, a task specification can include one or more declarations of ENTRY POINTS.

   c. For our next example, we will use a variant of the producer and consumer problem having three tasks: a Producer task, a Consumer task, and a Buffer task. All three tasks have separate specifications and bodies, but both are given inside the body of the overall procedure, rather than as separate compilation units.

      i. The producer task produces characters and stores them in the buffer. (In this case, the producer will read characters from a disk file).

      ii. The consumer task removes characters from the buffer and consumes them. (In this case, the consumer will write characters to a disk file in a different format)

      iii. The buffer task supports the other two. It has three entries

           - Store places a character in the buffer. (The Producer uses this)

           - Producer_Done notifies the buffer that the Producer is all through producing characters.

           - Retrieve obtains a character from the buffer. (The Consumer uses this)

      iv. Notice that the declaration of an entry point looks very much like the declaration of a procedure. In fact, from the standpoint of the task invoking an entry, the syntax is the same as for a procedure call.

2. A rendezvous is initiated when one task invokes an entry point of another task.

   a. Example: the following line occurs in the body of the Producer task:

      Buffer.Store(Card(I));

   b. However, unlike an ordinary procedure call, both tasks have to actively participate in a rendezvous. An entry call can only be completed if the recipient task has executed an accept statement for that entry. For example, the following appears in the body of the buffer task (where the buffer is implemented as a queue using a circular array.): [For now, ignore the select and when].

```
accept Store(C: in Character) do
    Buffer_Data(Buffer_Rear) := C;
    Buffer_Rear := (Buffer_Rear + 1) rem Buffer_Size;
    Buffer_Empty := False;
    Buffer_Full  := Buffer_Rear = Buffer_Front;
end Store;
```

   (Note that an accept statement can occur in the middle of the executable code of the accepting task.)

   c. Since a rendezvous can occur only with the cooperation of both tasks, it is possible that one task will have to wait for the other.

      i. If the producer task attempts to execute the Store rendezvous when the buffer is not ready to accept it, the producer task will be made to wait until the buffer is ready.

      ii. If the buffer task attempts to accept the Store rendezvous when no one is trying to do a store, the buffer task will be made to wait until someone does so.

   d. Of course, this raises a question in the case of a task like the buffer, which has several entry points. What if it tries to accept a Store entry when the producer has run out of cards and is trying to invoke the Producer_Done entry instead?

      i. The buffer will wait forever for an entry call that will never occur.

      ii. The producer will wait forever for the buffer to be ready to accept v, while it is hung elsewhere.

      iii. To deal with this kind of situation (which is very common), Ada provides a select statement whereby a task can announce its readiness to accept any one of several possible entries:

      Example: note the select statement in the Buffer task body, where there are 3 alternatives. The Buffer task will accept any one of them.

- If the select is executed when no entry call is pending, then the buffer task will wait for one to occur. If it is executed while one entry call is pending, then that rendezvous will occur at once. If it is executed when two or more entry calls are pending, then one rendezvous will occur - but which one it will be is not specified by the language - it could be any one of them.

iv. In this case, two of the entries are GUARDED - they are qualified by when clauses:

- the Store entry cannot be accepted when the buffer is full.

- the Retrieve entry cannot be accepted when the buffer is empty.

- (However, Producer_Done may be accepted at any time.)

- A guarded entry whose when condition is satisfied is said to be open, and one whose when condition is not satisfied is said to be closed. It is a run-time error to execute a select statement if all of its entries are closed, unless it has an else clause.

v. Not shown here is the possibility that a select statement may also include an else clause. In this case, if the select is executed and no entries are pending, then the else part is done immediately (the task does not wait.)

vi. In this respect, the Ada rendezvous is assymetric:

- A task that attempts an entry call which the destination task is not ready to accept will always be put into a wait state.

- A task that attempts to accept a rendezvous has considerable freedom, and need not be forced to wait for that specific rendezvous or even for any rendezvous at all.

3. With this background, we are ready to look at a complete implementation of the producer and consumer problem (with a buffer) in Ada.

   Go over complete program

E. Ada's concurrency facilities have been significantly extended in Ada95. However, we will not attempt to discuss these extensions here - mastering all the features of Ada (or any language) is beyond the scope of this course.