FlexOr Technology Series

Repack Implementation as Coroutines and Inversion

Gunnar Gotshalks
1999 August

Table of Contents

Introduction **1**

The Program Begins Here **2**
- Include base for the repack problem 2.1
- Base constant definitions 2.2
- Global objects for coroutine implementation 2.3

Coroutine implementation of the repack problem **3**
- Input process for the coroutine implementation 3.1
- Coroutine initialization of the input process 3.2
- Build a word 3.3
- Coroutine implementation of send a word 3.4
- Output process for the coroutine implementation 3.5
- Coroutine initialization of the output process 3.6
- Coroutine implementation of receive a word 3.7
- Supporting resume operation for the coroutine implementation 3.8
- Main program for the coroutine implementation 3.9

Inversion implementation of the repack problem **4**
- Macro definitions 4.1
- Input process for the inversion implementation 4.2
- Inversion restart of the input process 4.3
- Output process for the inversion implementation 4.4
- Inversion restart of the output process 4.5
- Main program, scheduler, for the inversion implementation 4.6

Make file for repack implementations **5**

1 Introduction

The repack problem is described in the *Monograph on Jackson System Development*. In this document we present two different implementations of the repack problem. The purposes in presenting two different implementations are as follows.

- Show a single design has more than one possible implementation. The choices typically are in how communication is effected among processes.
- Show how coroutines are implemented.
- Show how inversion is implemented.

2 The Program Begins Here

There are two subtrees to the program. The first subtree is a coroutine implementation of the repack problem. The second subtree is an inversion with respect to a scheduler implementation of the repack problem. Since so much of the implementation is the same many sections are used twice.

```
NEW OUTPUT FILE repackCr.c
    <Coroutine implementation of the repack problem 3>
NEW OUTPUT FILE repackInv.c
    <Inversion implementation of the repack problem 4>
NEW OUTPUT FILE Repack
    <Make file for repack implementations 5>
```

Program text is not referenced

2.1 Include base for the repack problem

The program uses standard library for input and output of characters.

```
#include <stdio.h>
```

Used in sections 3 and 4

2.2 Base constant definitions

Using C macros to define various constants used in the program.

```
#define EOLchar '\n'            /* End of line character */
#define EOSchar '\0'            /* End of string character */
#define MaxWordLength 40       /* Assume words have 40 or less characters. */
#define MaxOutputLineLength 40  /* Change for different output widths. */
```

Used in sections 3 and 4

2.3 Global objects for coroutine implementation

Coroutine implementation requires a savearea for the execution state (registers, stack information, etc.). This is provided in `jmp_buf`. There is one savearea for each coroutine plus one for the main program. The savearea for the main program is required to make use of `setjmp` and `longjmp`.

```
jmp_buf inputSavearea, outputSavearea, mainSavearea;
```

The message area `aWord` is global to both input and output coroutines.

```
char aWord[MaxWordLength];
```

Forward declarations of the processes and support operations to take care of the potential forward references.

```
void inputPhase();
void outputPhase();
void resume(char who);
```

Used in section 3

3 Coroutine implementation of the repack problem

Besides the base includes, we include the header file defining the operations `setjmp` and `longjmp`, and their associated data structures.

```
<Include base for the repack problem 2.1>
#include <setjmp.h>
<Base constant definitions 2.2>
<Global objects for coroutine implementation 2.3>
<Input process for the coroutine implementation 3.1>
<Output process for the coroutine implementation 3.5>
<Supporting resume operation for the coroutine implementation 3.8>
<Main program for the coroutine implementation 3.9>
```

Used in section 2

3.1 Input process for the coroutine implementation

Add a subprogram header. With coroutines there are no parameters. Declare the local variables as static so they remain in existence even though resuming their output operation interrupts the function. `pChar` is an additional implementation variable used to build a word.

```
void inputPhase()
{ static int aChar;
  static char *pChar;
```

Without having coroutines defined in the language we need to do extra work to set up the coroutine environment.

```
<Coroutine initialization of the input process 3.2>
```

The rest is a straightforward translation of the input process.

```
  aChar = getchar();
  while (aChar != EOF)
  { while (aChar != EOLchar)
    { switch (aChar)
      { case ' ' : aChar = getchar(); break;
        default  : <Build a word 3.3>
                   <Coroutine implementation of send a word 3.4>
      }
    }
  aChar = getchar();
  }
```

Send the null word at the end of input.

```
  aWord[0] = EOSchar;
  <Coroutine implementation of send a word 3.4>
```

After receiving the null word the output routine does not expect more input, so the input process will not be resumed. However, just in case the output process does resume the input process we have a guard return to the main program.

```
```

```
  longjmp(mainSavearea,0);
  }
```

Used in section 3

3.2 Coroutine initialization of the input process

The following statement is used to initialize the savearea for the input process. `setjmp` saves the current environment and returns 0, signaling that this is a return from a setjmp call. After initialization we return to the main program `longjmp(mainSavearea,0)` so it can initialize the output phase.

A resume occurs when `longjmp(inputSavearea,0)` is used. This simulates a return from `setjmp` shown below. We pass a value 0 to `longjmp` which in turn causes `setjmp` to return 1 −− weird Sun implementation of `setjmp`and `longjmp`. Thus, in the if statement a return from `longjmp` to the input process causes the process to continue as if it was not interrupted.

```
  if (setjmp(inputSavearea)==0) longjmp(mainSavearea,0);
```

Used in section 3.1

3.3 Build a word

```
pChar = aWord;                /* word <- '' */
while ((aChar != EOLchar) && (aChar != ' '))
{ *pChar++ = aChar;           /* word <- word || aChar */
  aChar = getchar(); }
*pChar = EOSchar;
```

Used in sections 3.1 and 4.2

3.4 Coroutine implementation of send a word

Similar semantics as in the section "Coroutine initialization of the input process" except that after setting the environment we resume the output process. Because the compiler does not support the resume instruction, we fake it with a custom `resume` operation and pass the "name" of the routine to resume. We do not have to transfer the word into the common message area because the common message area was used to build the word −− in effect the transfer is implicit.

```
if (setjmp(inputSavearea)==0) resume ('B');
```

Used in sections 3.1 and 3.1

3.5 Output process for the coroutine implementation

Add a subprogram header. With coroutines there are no parameters. Declare the local variables as static so they remain in existence even though resuming the the input operation interrupts the function.

```
void outputPhase()
{ static int outputLineLength;
```

Without having coroutines defined in the language we need to do extra work to set up the coroutine environment.

<Coroutine initialization of the output process 3.6*>*

The rest is a straight−forward translation of the input process. There is no need to transfer the word from the message buffer in this program. We output directly from the buffer.

```
  while (strlen(aWord) != 0)
  { printf("%s",aWord); outputLineLength = strlen(aWord);
    <Coroutine implementation of receive a word 3.7>
    while (strlen(aWord) != 0
          && strlen(aWord) + 1 + outputLineLength <= MaxOutputLineLength)
    { printf(" %s",aWord); outputLineLength += strlen(aWord) +1;
      <Coroutine implementation of receive a word 3.7>
      }
    printf("\n"); outputLineLength = 0;
    }
  if (outputLineLength != 0) printf("\n");
```

When the last word is output, return to the main program. There cannot be any more input so the input process does not need to be resumed.

```
  longjmp(mainSavearea,0);
  }
```

Used in section 3

3.6 Coroutine initialization of the output process

See the section "Coroutine initialization of the input process" for an explanation of what is happening. This section also takes care of the first receive word in the process.

```
  if (setjmp(outputSavearea)==0) longjmp(mainSavearea,0);
```

Used in section 3.5

3.7 Coroutine implementation of receive a word

See the comments in section "Coroutine implementation of send a word". Receiving is analogous to sending.

```
if (setjmp(outputSavearea )== 0) resume('A');
```

Used in sections 3.5 and 3.5

3.8 Supporting resume operation for the coroutine implementation

To make `longjmp` work correctly, it is necessary to have something on the runtime stack for `longjmp` to pop. Thus, instead of just long jumping between coroutines we call a subprogram that does the long jump for us. The structure is easily extended to more coroutines.

```
void resume(char who)
{ switch (who)
  { case 'A': longjmp(inputSavearea,0); break;
    case 'B': longjmp(outputSavearea,0); break;
    }
  }
```

Used in section 3

3.9 Main program for the coroutine implementation

The main program sets up the environments for the coroutines and starts the appropriate coroutine. The structure is easily extended to more coroutines.

```
int main()
{
```

Start up both processes with their first call.

```
  if (setjmp(mainSavearea) ==0 ) inputPhase();
  if (setjmp(mainSavearea)== 0 ) outputPhase();
```

Start the first coroutine.

```
  if (setjmp(mainSavearea) == 0) resume('A');
  return 0;
  }
```

Used in section 3

4 Inversion implementation of the repack problem

```
<Include base for the repack problem 2.1>
<Base constant definitions 2.2>
<Macro definitions 4.1>
<Input process for the inversion implementation 4.2>
<Output process for the inversion implementation 4.4>
<Main program, scheduler, for the inversion implementation 4.6>
```

Used in section 2

4.1 Macro definitions

Define a macro for interrupting an inverted process −− simulating the send and receive operations. Every send and receive operation is replaced with a macro invocation using a different label every time and an appropriate reason for the interrupt.

```
#define interruptTheProcess(label,reason)  *reasonForNextCall = reason;\
          nextStatement = label; return ; L##label :
```

Create an abbreviation for the input call reason list.

```
typedef enum inputCallReasonList {inputDone, sendRepack} inputCallReason;
```

Create an abbreviation for the output call reason list */

```
typedef enum outputCallReasonList {outputDone, receiveRepack} outputCallReason;
```

Used in section 4

4.2 Input process for the inversion implementation

Define the input process subprogram header with one parameter per inverted channel, the word channel, and one for the reason for the next call.

```
void inputPhase(char *chanWord, inputCallReason *reasonForNextCall)
```

The variables that must remain unchanged from call to call are declared to be static so they are not created on the stack on every call but rather remain defined until the entire program terminates.

```
{ static int aChar;
  static char aWord[MaxWordLength], *pChar;
```

Add the new variable nextStatement to keep track of the statement from which execution begins on the next call.

```
  static int nextStatement = 0;
  <Inversion restart of the input process 4.3>
```

A straightforward translation of the input process.

```
  aChar = getchar();
  while (aChar != EOF)
  { while (aChar != EOLchar)
    { switch (aChar)
      { case ' ' : aChar = getchar(); break;
        default  : <Build a word 3.3>
```

Unlike coroutines each send operation is customized because of the need for a unique label and reason for next call. Notice that the interrupt is before the actual send (putting the word into the parameter). The program must return to the caller with the information that the next call will transmit the word, so the scheduler will know when to call the routine.

```
                   interruptTheProcess(1,sendRepack)
                   pChar = aWord;
                   while (*pChar != EOSchar) *chanWord++ = *pChar++;
                   *chanWord = EOSchar;
```

Continue with the translation.

```
        }
      }
    aChar = getchar();
    }
```

Send the null word at the end of input.

```
  interruptTheProcess(2,sendRepack)
  *chanWord = EOSchar;
```

Always have a done reason at the end. It not only makes for a robust routine, but also is necessary to make sure the last send is done.

```
  interruptTheProcess(3,inputDone)
  return;
```

```
}
```

Used in section 4

4.3 Inversion restart of the input process

The first executable statement in an inverted process is a jump to the point of the last interruption. In C we need a switch statement to map from integer labels to program labels.

```
switch(nextStatement)
{ case 0: goto L0;
  case 1: goto L1;
  case 2: goto L2;
  case 3: goto L3;
  }
```

Always add a label L0 at the beginning of the inverted process to handle the initial call.

```
L0:
```

Used in section 4.2

4.4 Output process for the inversion implementation

Define the output process subprogram header with one parameter per inverted channel, the word channel, and one for the reason for the next call.

```
void outputPhase(char *aWord, outputCallReason *reasonForNextCall)
```

The variables that must remain unchanged from call to call are declared to be static so they are not created on the stack on every call but rather remain defined until the entire program terminates.

```
{ static int outputLineLength;
```

Add the new variable nextStatement to keep track of the statement from which execution begins on the next call.

```
static int nextStatement = 0;
<Inversion restart of the output process 4.5>
```

A straightforward translation of the input process. We use the data in the parameter directly, i.e. do not copy to a local variable, because there is no interrupt until the word is processed.

```
interruptTheProcess(1,receiveRepack)

while (strlen(aWord) != 0)
{ printf("%s",aWord); outputLineLength = strlen(aWord);
  interruptTheProcess(2,receiveRepack)
  while (strlen(aWord) != 0
        && strlen(aWord) + 1 + outputLineLength <= MaxOutputLineLength)
  { printf(" %s",aWord); outputLineLength += strlen(aWord) +1;
    interruptTheProcess(3,receiveRepack);
    }
  printf("\n"); outputLineLength = 0;
  }
```

Always terminate an inverted process with an interrupt reason "done". It makes for a robust operation.

```
  interruptTheProcess(4,outputDone);
  return;
  }
```

Used in section 4

4.5 Inversion restart of the output process

The first executable statement in an inverted process is a jump to the point of the last interruption. In C we need a switch statement to map from integer labels to program labels.

```
  switch(nextStatement)
  { case 0: goto L0;
    case 1: goto L1;
    case 2: goto L2;
    case 3: goto L3;
    case 4: goto L4;
    }
```

Always add a label L0 at the beginning of the inverted process to handle the initial call.

```
L0:
```

Used in section 4.4

4.6 Main program, scheduler, for the inversion implementation

The main program is the scheduler for the two processes. In such a simple problem as the repack problem the scheduler is trivial because there is no choice. All that can be done is get from the sender and give to the receiver.

```
int main()
{ char aWord[MaxWordLength];
  inputCallReason reasonForNextInputCall;
  outputCallReason reasonForNextOutputCall;
```

Start up both processes with their first call.

```
  inputPhase(aWord, &reasonForNextInputCall);
  outputPhase(aWord, &reasonForNextOutputCall);
```

Loop until there are no more words being sent by the input phase, this is forward chaining. One could alternately loop until the output phase is done, which is backward chaining.

```
  while (reasonForNextInputCall != inputDone)
  { inputPhase(aWord, &reasonForNextInputCall);
    outputPhase(aWord, &reasonForNextOutputCall);
   }
  return 0;
  }
```

Used in section 4

5 Make file for repack implementations

```
# A primitive makefile to test the repack problem implementations.
# NOTE: The program assumes all line endings are Unix style.  It fails on
# a bus error on Mac endings.  Dos line endings may work because they have
# a line feed by the carriage return may cause display/print problems (it has
# not been tested.
#
# First run "java FlexOr.sgml.Start" and tangle a copy of the file
# repack.sfx.
#
# To compile all programs use "make -f Repack all" or "make -f Repack"
# To compile the coroutine version of repack use "make -f Repack repackCr"
# To compile the inverted version of repack use "make -f Repack repackInv"
#
# To execute the programs use io redirection as in the following.
#        repackCr  < inputFile > outputFile
#        repackInv < inputFile > outputFile
# The gcc compiler (GNU C) is used because it gives better error messages
# than the cc compiler.

CC      = gcc
LINKER  = gcc

all: repackCr repackInv

repackCr: repackCr.o
 $(LINKER) -o $@ $(LDFLAGS) repackCr.o

repackInv: repackInv.o
 $(LINKER) -o $@ $(LDFLAGS) repackInv.o
```

Used in section 2