CORE EXAM: SYSTEMS PART

Fall 2006
(Version with Answers)

PL&C: Question 1

Consider the following piece of C code:

```
int intsqrt()
{
  int n = 500;
  int y = 0;
  int w = 1;
  while (n >= w) {
    w += 2*(y++) + 3;
  }
  return y;
}
```

1. Give three-address code for the body of the function intsqrt.

   Answer:

   ```
   1. n := 500
   2. y := 0
   3. w := 1
   4. if n < w goto 10
   5. t1 := y * 2
   6. t2 := t1 + 3
   7. w := w + t2
   8. y := y + 1
   9. if n >= w goto 5
   10.
   ```

2. Suppose your code is processed by an optimizing compiler that implements common subexpression elimination, copy propagation, dead-code elimination, code motion, induction variable elimination, and strength reduction. Give the resulting optimized three-address code.

   Answer:

   After copy propagation:

   ```
   1. n := 500
   2. y := 0
   3. w := 1
   4. if 500 < 1 goto 10
   ```

```
5. t1 := y * 2
6. t2 := t1 + 3
7. w := w + t2
8. y := y + 1
9. if 500 >= w goto 5
10.
```

After dead-code elimination:

```
1. y := 0
2. w := 1
3. t1 := y * 2
4. t2 := t1 + 3
5. w := w + t2
6. y := y + 1
7. if 500 >= w goto 3
8.
```

After strength reduction:

```
1. y := 0
2. w := 1
3. t1 := 0
4. t2 := t1 + 3
5. w := w + t2
6. y := y + 1
7. t1 := t1 + 2
8. if 500 >= w goto 4
9.
```

After induction variable elimination:

```
1. w := 1
2. t1 := 0
3. t2 := t1 + 3
4. w := w + t2
5. t1 := t1 + 2
6. if 500 >= w goto 3
7. y := t1 / 2
```

3. How many assembly instructions will be executed if your code is compiled and run?

   Answer:

   Each three-address instruction corresponds to a single assembly instruction. Lines 3 through 6 get executed 22 times while the others get executed only once each. Thus, the total number of instructions is 91.

PL&C: Question 2

1. Write a function called *rev* in ML or Scheme that takes as input a list L and returns a list consisting of the same elements as L but in reverse order.

   Answer:

   Scheme:

   ```
   (define (rev L)
           (cond ((null? L) ())
                 (else (append (rev (cdr L)) (list (car L))))))
   ```

   ML:

   ```
   fun rev [] = []
     | rev x::t = (rev t) @ [x];
   ```

2. Suppose that *list.h* consists of the following C++ code:

```
template <class T>
class List {
  class ListElem {
  public:
    T _data;
    ListElem* _next;
    ListElem(const T& data, ListElem* next) : _data(data), _next(next) {}
  };
  ListElem* _top;
public:
  List() : _top(NULL) {}
  ~List();
  void insert(const T& data) { _top = new ListElem(data, _top); }
  void reverse();
};
```

Write the code for the function `reverse` as it would appear in a file that includes *list.h*. The function should reverse the elements in the list without allocating or freeing any additional memory.

Answer:

```
template <class T>
void List<T>::reverse()
{
  if (_top == NULL) return;
  ListElem* prev;
  ListElem* next = _top->_next;
  _top->_next = NULL;

  while (next) {
    prev = _top;
    _top = next;
    next = _top->_next;
    _top->_next = prev;
  }
}
```

PL&C: Question 3

Consider the alphabet $\Sigma_1 = \{0, 1, +, \times\}$, where we refer to the letters 0 and 1 as *operands* and the symbols $+$ and $\times$ as *operators*. A well-formed arithmetic expression is a string whose first and last symbols are operands and, within the string, operands and operators alternate.

A) (3 points) Construct a finite-state automaton that accepts the language of all well-formed arithmetic expressions.

Answer
The automaton is defined as follows:

- States: $\{q_0, q_1\}$.
- Initial state: q_0.
- Accepting state: q_1.
- Transition function defined by the following table:

	$\{0,1\}$	$\{+,\times\}$
q_0	q_1	$-$
q_1	$-$	q_0

B) (3 points) Construct a finite-state automaton that accepts the language of all arithmetic expressions which evaluate to a positive (non-zero) value. Note that $\times$ has higher priority than $+$. Thus, $0 + 1 \times 0$ should be interpreted as $0 + (1 \times 0)$ rather than $0 + (1 \times 0)$. You may present your automaton either as a graph with nodes representing the states and edges labeled by letters, or in a tabular form.

Answer
An NFA (non-deterministic automaton) which recognizes the language of positive expressions is presented in Fig. 1. Note that the initial states are $\{q_0, q_2\}$, and the accepting states are $\{q_3, q_5\}$.

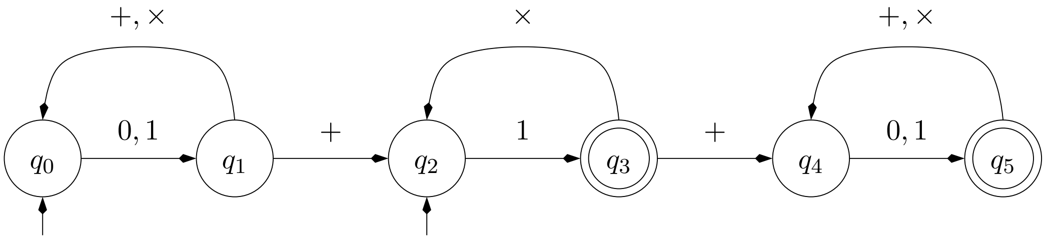

Figure 1: Automaton for Question B

C) (4 points) Extend the alphabet to $\Sigma_2 : \{0, 1, +, \times, (,)\}$. That is, allow parenthesis within the arithmetic expressions. Construct a push-down automaton that accepts all proper arithmetic expressions over Σ_2 evaluating to 0. An arithmetic expression is proper if its parentheses structure is balanced.

Answer
As a first step, we construct a context-free grammar which generates the language of arithmetical expressions over Σ_2 whose value is 0. In this grammar the start symbol is E_0.

$$
\begin{array}{lcl}
E & \rightarrow & T \quad | \quad E + T \\
T & \rightarrow & P \quad | \quad T \times P \\
P & \rightarrow & 0 \quad | \quad 1 \quad | \quad (E) \\
E_0 & \rightarrow & T_0 \quad | \quad E_0 + T_0 \\
T_0 & \rightarrow & P_0 \quad | \quad T_0 \times P \quad | \quad T \times P_0 \\
P_0 & \rightarrow & 0 \quad | \quad (E_0)
\end{array}
$$

Next, we use the standard construction which builds a non-deterministic push-down automaton which recognizes the language generated by the above grammar. The automaton is presented in Fig. 2. A word is accepted iff it may cause the automaton to reach the accepting state q_2.

State	Stack Top	Input	New State	Stack Replacement
q_0	$-$	ϵ	q_1	$E_0 Z$
q_1	E	ϵ	q_1	T
q_1	E	ϵ	q_1	$E + T$
q_1	T	ϵ	q_1	P
q_1	T	ϵ	q_1	$T \times P$
q_1	P	0	q_1	$-$
q_1	P	1	q_1	$-$
q_1	P	(	q_1	$E)$
q_1	E_0	ϵ	q_1	T_0
q_1	E_0	ϵ	q_1	$E_0 + T_0$
q_1	T_0	ϵ	q_1	P_0
q_1	T_0	ϵ	q_1	$T_0 \times P$
q_1	T_0	ϵ	q_1	$T \times P_0$
q_1	P_0	0	q_1	$-$
q_1	P_0	(	q_1	$E_0)$
q_1	$+$	$+$	q_1	$-$
q_1	$\times$	$\times$	q_1	$-$
q_1	)	)	q_1	$-$
q_1	Z	ϵ	q_2	$-$

Figure 2: Push-down automaton for Question C

OS: Question

1 Operating Systems: Concurrency

Many applications, notably network servers, require concurrency in that they can perform many different activities, such as processing different requests, more or less at the same time.

1.1 The Thread Abstraction

A popular abstraction for expressing concurrency are *threads* of execution, running within the same process.

- In one concise sentence, what is the conceptual facility provided by a thread?

  Answer: A thread effectively virtualizes the CPU, providing the illusion that the thread owns the CPU.

- In one concise sentence, the threading package (i.e., the code implementing the thread abstraction) needs to keep what state *per* thread?

  Answer: The stack, register set (including program counter), and any internal book-keeping information.

- In one concise sentence, how does this compare to a traditional Unix process?

  Answer: A traditional Unix process also virtualizes memory, while threads share the same address space.

1.2 User-Level v Kernel-Level Threads

The threading package can be implemented at the user-level or the kernel-level (or both).

- For any common thread operation, such as acquiring a lock, how many system calls have to be executed for a user-level implementation? Please limit your answer to one word or number.

  Answer: None.

- For any common thread operation, such as acquiring a lock, how many system calls have to be executed for a kernel-level implementation? Please limit your answer to one word or number.

  Answer: One.

- Based on the previous two answers and in one concise sentence, which implementation is likely to perform better and why?

  Answer: User-level threads are likely to be faster because they do not require system calls.

Imagine an application with two threads `S` and `T`, with `S` currently running and `T` leisurely lounging on the ready queue. Now, imagine that thread `S` initiates an I/O operation, such as `fwrite()` or `fflush()`.

- For a user-level threading package implementation, what will happen to `T`? Please limit your answer to one concise sentence.

  Answer: Nothing, as the kernel has no knowledge of the user-level ready queue.

- For a kernel-level threading package implementation, what will happen to `T`? Please limit your answer to one concise sentence.

  Answer: The threading package will schedule thread `T` after moving `S` onto the busy queue.

- Based on your previous two answers and in one concise sentence, which implementation is likely to perform better in the presence of frequent I/O operations, such as `fwrite()` or `fflush()`, and why?

  Answer: Kernel-level threads are likely to perform better because the scheduler knows which threads are blocking on I/O *and* which threads are ready to execute.

1.3 Events

Event-based programming provides an alternative to threads that can avoid some of the limitations/trade-offs explored in the previous questions. However, to support event-based programming, the interaction between applications and the kernel needs to be changed as well. Notably, I/O operations need to be *asynchronous*, i.e., return as soon as possible and not wait for completion of the I/O. Furthermore, the kernel needs to provide a new system call, such as `select()`, that notifies applications of completed I/O operations. Finally, for each received notification, i.e., *event*, the application needs to run the appropriate code, i.e., *event handler*.

- Kernel notifications may arrive faster than the event handlers complete their processing. Consequently, the application's event package needs to keep what state?

  Answer: A queue of pending event handlers.

- In one concise sentence, how does this state compare with that of a threading package?

  Answer: There's less (no stack) and it's not as low level (no registers).

- Consider the following sequence of (abstract, high-level) operations in a threaded system:

```
void sequence() {
  compute1();
  read();
  compute2();
  write();
  compute3();
}
```

  How would this sequence be implemented in an event-based system? Use the same C-like function notation as above, making all event handlers explicit and using descriptive names for each operation.

  Answer:

```
void sequence_part1() {
  compute1();
  start_read();
}
```

```
void sequence_part2() {
  compute2();
  start_write();
}

void sequence_part3() {
  compute3();
}
```

- Based on your previous answer and in one concise sentence, why is event-based programming often considered harder than threaded programming?

  Answer: Since sequences of operations performing I/O need to be broken into several functions, the logical flow of a program is more obscure than for threaded programs.