1 Assumptions and Definitions

For the exercises we will consider, assume the following about the computer architecture: (these assumptions appear in Gregory R. Andrews text, "Foundations of Multithreaded, Parallel and Distributed Programming")

- Values of the basic types (e.g., int) are stored in memory elements (e.g., words) that are read and written as atomic actions.

- Values are manipulated by loading them into registers, operating on them there, then storing the results back into memory.

- Each process has its own set of registers. This is realized either by having distinct sets of registers or by saving and restoring register values whenever a different process is executed. (This is called a context switch since the registers constitute the execution context of a process.)

- Any intermediate results that occur when a complex expression is evaluated are stored in registers or in memory private to the executing process—e.g., on a private stack.

Based on the above assumptions, we have the following definitions:

**critical reference** In an expression, a critical reference is a reference to a variable that is changed by another process.

**At-Most-Once Property** An assignment statement $x = e$ satisfies the at-most-once property if either

1. $e$ contains at most one critical reference and $x$ is not referenced by another process, or
2. $e$ contains no critical references, in which case $x$ may be read by other processes.
The At-Most-Once Property for assignment statements is so-called because there can be at most one shared variable, and it can be referenced at most one time. A similar definition exists for expressions not contained in assignment statements: an expression satisfies the At-Most-Once Property if it contains no more than one critical reference.

2 Atomicity: Keeping up Appearances

Why do we care about the At-Most-Once Property? Because assignment statements with this property will *appear* to execute atomically. Why? Here are two examples. First:

```c
int x = 0, y = 0;
c o x = x+1; // y = y+1; oc;
```

What are the final values of x and y? And second:

```c
int x = 0, y = 0;
c o x = y+1; // y = y+1; oc;
```

What are the final values of x and y?

**Lesson:** the appearance of atomicity in assignment statements does not necessarily mean deterministic runtime behavior.

Now let’s see an example where the At-Most-Once Property does *not* hold:

```c
int x = 0, y = 0;
c o x = y+1; // y = x+1; oc;
```

Some questions:

• Why doesn’t the At-Most-Once Property hold for either assignment statement?

• There are more possibilities for the final values of x and y. What are they?

• Why do these assignments no longer appear to execute atomically? For which final values?

3 Atomicity: Reducing the Possibilities

When we can no longer satisfy the At-Most-Once Property, we must often resort to specifying more course-grained levels of atomicity. Recall the three varieties of the *await* statement:

1. `<await (B) $S;$>` — mutual exclusion and condition synchronization

2. `<$S;$>` — mutual exclusion only

3. `<await (B);$>` — synchronization only
4 Safety and Liveness Properties

Recall that a property is a predicate that is true for all possible histories of a program’s execution. That is, even in the presence of nondeterminism, a property remains true for all possible runs of a program.

Recall there are two broad categories of properties: safety and liveness. Here are the general definitions, once more:

**safety property**  Assertion that nothing bad ever happens during a program’s execution.

**liveness property**  Assertion that something good eventually happens during a program’s execution.

Examples of safety properties include mutual exclusion and deadlock freedom. Badness for mutual exclusions occurs when more than one process enters a critical section of code at the same time. Deadlock is the condition where all processes in a computation are blocked waiting for a condition that will never change, precisely because all the processes that could change it are blocked.

Examples of liveness properties include eventual entry into a critical section, that the request for a service (e.g., printing, reading from disk) will eventually be granted, and delivery of messages sent. Liveness properties are affected by scheduling policies.

5 Scheduling Policies and Fairness

A scheduling policy determines which of a set of eligible atomic actions executes next. Scheduling policies need not be fair, and in fact there are degrees of fairness. An eligible process that is never scheduled to execute is said to starve. Three main types of fairness are:

**Unconditional Fairness**  every unconditional atomic action that is eligible is executed eventually.

**Weak Fairness**  A scheduling policy is weakly fair if:

1. it is unconditionally fair, and
2. every conditional atomic action that is eligible is executed eventually, assuming that its condition becomes true and then remains true until it is seen by the process executing the conditional atomic action.

**Strong Fairness**  A scheduling policy is strongly fair if:

1. it is unconditionally fair, and
2. every conditional atomic action that is eligible is executed eventually, assuming that its condition is infinitely often true.

Some comments on the three types of fairness. Recall the three types of \texttt{await} statements. Unconditional fairness is fine for programs with mutual exclusion only (i.e., \texttt{<S;>}) statements; but for programs with conditional atomic actions, we desire more from the scheduling policy.

Consider the first version of the \texttt{await} statement. If \texttt{<await (B) S;>} is eligible and \texttt{B} becomes true, then \texttt{B} remains true at least until after the conditional atomic action has been executed. This is weak fairness. Examples include time-slicing and round-robin. In these cases, each delayed process will see that its delay condition becomes true.

But weak fairness doesn’t guarantee that any eligible \texttt{await} statement eventually executes. Consider the case where the condition \texttt{B} changes from false to true, and back to false, while the eligible process is delayed. For this possibility, we need a stronger scheduling policy: Strong Fairness.

In the definition of strong fairness, what is meant by infinitely true refers to the existence of a cycle in a nonterminating program, where during certain periods of time the condition is true, and during other periods the condition is false. A strongly fair scheduler must give an eligible process the chance, over time, to execute during both possible periods.

To see the difference between weak and strong fairness, consider this example:

\begin{verbatim}
bool continue = true, try = false;
co while (continue) { try = true; try = false; }
// <await (try) continue = false;>
\end{verbatim}

With a strongly fair policy, this program will eventually terminate, because \texttt{try} is infinitely often true. However, with a weakly fair policy, the program might not ever terminate, because \texttt{try} is also infinitely often false.

\textbf{Depressing reality:} it is impossible to devise a processor scheduling policy that is both \textit{practical} and strongly fair. Why?