1 Introduction

One method to synchronize multiple processors is Mutual Exclusion, as we saw in the last class. However, that often is not efficient, because it serializes at least the code in the Critical Section, and perhaps more.

Serializing a large piece of code is less efficient than serializing small pieces. Last time, we saw the example of finding all primes from 1 to $N$, for some $N$. The naive way to solve this problem would be to split the interval $[1, \ldots, N]$ into $k$ equal pieces, and give one piece to each of $k$ processors. But that is not best, because each piece may require a different amount of processing time. So we need an algorithm of higher granularity: the numbers $1, \ldots, k$ go to processors 1 through $k$, and as processors finish, they take the first number not yet assigned. Only the piece of code that determines which new number to assign needs to be synchronized.

2 Amdahl’s Law

In the ideal situation, $n$-way parallelism would provide $n$-fold speedup.

If the problem is independently dividable, the threads don’t need to communicate. Problems in scientific computing often have this feature. But in pure theory we can talk about inherent limitations on parallelizing problems.

Amdahl’s Law For a given task, let $P$ be the fraction of work that is parallelizable; and let $S$ be the ratio between the time for one processor to complete the task, vs. the time taken by $n$ processors. Then

$$ S = \frac{1}{\frac{P}{n} + (1 - P)}. $$

The total time needed to complete the parallelizable part of the task is $P/n$, while $(1 - P)$ is the time needed to complete the part that is not parallelizable.

3 Example: painters painting rooms

Suppose there are five painters painting a five-room house. Four rooms are the same size, but the fifth room is double the size. Then $P = 5/6$, as there are six “units” to paint. We assume
there is no communication between the painters. As a result there is low efficiency (nonideal speedup), because the first four wait around, not knowing the fifth painter needs help finishing the last room. More precisely:

\[
S = \frac{1}{\frac{5}{6} + (1 - \frac{5}{6})} = 3
\]

and \(3 < 5\) (where 5 is ideal).

Now suppose 10 painters are painting 10 rooms, and one of those rooms is double size. Then \(P = 10/11\), so

\[
S = \frac{1}{\frac{10}{11} + \frac{1}{11}} = 5.5
\]

and \(5.5 < 10\). The ratio between actual and ideal speedup worsens because more painters sit idle as the last painter finishes.

### 4 Mutual Exclusion, revisited

Mutual Exclusion is one possible way to allow coordination. This is sometimes called using “locks.” We can also handle coordination without locks. One advantage of Mutual Exclusion locks is that once you assume your ME algorithm works, there isn’t much else to argue about for correctness. Without locks, basic steps in your algorithm can happen concurrently, and proofs can be difficult, because there may be weird or invalid things going on, due to concurrency. However, algorithms without locks are much more efficient, and are more fault tolerant.

In particular, Mutual Exclusion can cause complete system failure, if the processor everyone is waiting for fails. So we are going to explore wait-free algorithms later in this course. Today, though, we will continue with Mutual Exclusion.

#### 4.1 Peterson’s Algorithm

Any Mutual Exclusion algorithm has four different regions:

1. ENTRY: processor taking steps to enter CS
2. CS: critical section
3. EXIT: code the processor executes once it leaves CS
4. REM: The remainder of the processor’s code.

This is the same algorithm we discussed in the previous class, though the text is slightly different. It satisfies No Deadlock, but not No Starvation.
Algorithm 1 Peterson’s Algorithm: Algorithm for processor $p_0$

\begin{align*}
\langle \text{ENTRY}\rangle & \quad \text{want}_0 := 1 \\
\text{repeat} & \\
\quad & \quad \text{wait} \\
\text{until} & \quad \text{want}_1 = 0 \\
\langle \text{CS}\rangle & \\
\langle \text{EXIT}\rangle & \quad \text{want}_0 := 0 \\
\langle \text{REM}\rangle & 
\end{align*}

Algorithm 2 Peterson’s Algorithm: Algorithm for processor $p_1$

\begin{align*}
\langle \text{ENTRY}\rangle & \quad \text{L: want}_1 := 0 \\
\text{repeat} & \\
\quad & \quad \text{wait} \\
\text{until} & \quad \text{want}_0 = 0 \\
\text{want}_1 & := 1 \\
\text{if} & \quad \text{want}_0 = 1 \ \text{then} \\
\quad & \quad \text{goto L} \\
\text{end if} \\
\langle \text{CS}\rangle & \\
\langle \text{EXIT}\rangle & \quad \text{want}_1 := 0 \\
\langle \text{REM}\rangle & 
\end{align*}
4.2 Peterson’s Algorithm with a priority bit

We are looking now at algorithms on atomic R/W registers. We add a priority bit to Peterson’s Algorithm, in order to obtain a Mutual Exclusion algorithm that satisfies No Starvation. Since the combined algorithms in Section 4.1 satisfy Mutual Exclusion, this one does too, as \( p_0 \) and \( p_1 \) never have the priority bit at the same time. There are three shared variables in this algorithm. We can base an \( n \)-processor algorithm on this by using a tournament tree, as we will see below.

To prove this algorithm satisfies No Deadlock, assume there is a processor waiting to enter CS, but, even so, no processor ever enters CS. There are two cases: (1) one processor is waiting, or (2) both processors are waiting. Eventually, only one of three things could happen.

1. A processor waits forever at the first loop
2. A processor waits forever at the second loop
3. A processor never gets past the second if statement

The proof would show that, as expected, none of these cases can happen.

4.3 Multiprocessor algorithm

We now extend Peterson’s Algorithm to multiple processors by using a tournament tree.
Consider Figure 1. We run the two-processor Peterson algorithm at each node. Whichever processor enters CS gets to advance to the next level of the tree. Suppose Mutual Exclusion is not satisfied at Node 1. Then there must be more than one processor entering Node 1 from either Node 2 or Node 3. So Mutual Exclusion does not hold at either Node 2 or Node 3, so it must not have held at one of their child nodes. By contradiction, we show that ME must hold at Node 1.

Alternatively, we could argue by induction that Mutual Exclusion is satisfied at each level, so must be satisfied at the top.

Assume there is Deadlock. Then there must be Deadlock at some node, but each two-processor algorithm satisfies No Deadlock, so each node of the tree satisfies No Deadlock also. A similar argument shows that this multiprocessor algorithm satisfies No Starvation.

Note that we need $O(n)$ shared bits for this algorithm. More generally, we need linear bits for any Mutual Exclusion algorithm.

## 5 Lower bound on Mutual Exclusion algorithms

We will now show that the lower bound on the number of shared R/W variables in a Mutual Exclusion algorithm for $n$ processors is $n$.

We make a simplifying assumption: we suppose all variables are single-writer. It is then intuitively reasonable that every processor needs to indicate in some manner whether it is entering CS. Suppose there are fewer than $n$ registers. Then some $p_i$ cannot write at all. But then there are two system states $s_0$ and $s_1$ and a processor $p_j$ such that $s_0 \sim s_1$ ($p_j$ cannot tell the difference between $s_0$ and $s_1$) such that $p_i$ is outside of CS in one state, but inside CS in
Figure 2: Since $p_i$ cannot write, processor $p_j$ cannot tell the difference between states $s_0$ and $s_1$, so it will execute schedule $\sigma_1$ from either state. But if $p_j$ executes $\sigma_1$ from $s_1$, then both $p_i$ and $p_j$ end up in the Critical Section, contradicting Mutual Exclusion.

Without the simplifying assumption, every processor should be able to write sometime, somewhere. Suppose $p_i$ writes to register $r_i$, and $p_j$ writes to $r_j$. If $r_i = r_j$, $p_j$ can erase information from $p_i$, and not know whether $p_i$ has entered CS. So like the previous (two-processor) case, we either have Deadlock, or risk two processors entering CS at the same time.

The flaw in this argument is that there may be other writes. To argue that all writes obliterate information, we need a nontrivial inductive argument. For a process to enter CS, it should not be able to distinguish its state from a state in which all other processors are in the remainder section.

Finally, note that we could avoid this situation in a synchronous system.

6 Lamport’s Bakery Algorithm

Lamport’s Bakery Algorithm has an advantage over the Peterson algorithm, in that it is FIFO: processors enter CS in the order they request it. The problem with this algorithm is that the values in the registers are potentially unbounded (if there are no quiescent periods). There is a nontrivial way to bound these values, by assigning tickets out of a cycle of values.
Algorithm 4 Lamport’s Bakery Algorithm

CHOSE\([i]\) := 1
NUM\([i]\) := max(NUM[1], \ldots, NUM[n]) + 1
CHOSE\([i]\) := 0
for \(j := 1\) to \(n\) do
  repeat
    wait
  until CHOSE\([j]\) = 0
  repeat
    wait
  until NUM\([j]\) = 0 or (NUM\([j]\), ID\(_j\)) ≥ (NUM\([i]\), ID\(_i\))
end for
⟨CS⟩
⟨EXIT⟩ NUM\([i]\) := 0
⟨REM⟩