ECE3140 / CS3420
Embedded Systems

Concurrency Basics

Prof. José F. Martínez

Cornell University

Outline: Concurrency

- Definition and challenge
- Basic assumptions
  - Private vs. shared memory
  - Atomicity
- Execution traces
  - Multiple possible executions
- Mutual exclusion
  - Properties
  - Algorithms
- Reference
  - Dijkstra's lecture note: E.W.Dijkstra Archive: Cooperating sequential processes (EWD 123)
    - Blackboard: Content → Resources

Concurrency

- What do we mean by "concurrent"?
  - . . . running in parallel, operating at the same time. (Webster)
  - . . . existing or acting together or at the same time. (Oxford)

- Multiple programs may run concurrently through context switching or on multiple cores

- Challenge:
  - Operations from concurrent programs may be interleaved in many different ways, and lead to non-deterministic outcomes

- For the moment:
  - Avoid the assumptions on physical time
  - Think about how different operations are ordered

Shared Memory

- How do two programs (a.k.a. process) communicate?
- In shared memory systems, multiple programs communicate through shared memory
  - Program P1 (sender) writes to a shared memory location
  - Program P2 (receiver) reads from the memory location
- Classify variables into two kinds:
  - **Shared variables**: those accessed by more than one process
  - **Private variables**: those accessed by one process
- Process vs. threads
  - Process: a running program often with its own memory space
  - Thread: an independent execution within a process, with shared memory space; a process has one or more threads.
  - In this discussion, we will primarily use the term ‘process’ to refer to multiple concurrent programs with shared memory

Basic Assumptions

- Non-interference:
  - the concurrent activities of program parts that do not share variables do not interfere with each other.

- Atomicity
  - a single read or a single write to a shared variable is an indivisible (atomic) action.

- It is important to note these are assumptions!
  - Assignments cannot “collide” to produce a different result
  - This is a requirement of the implementation—it is not free!

Atomicity

- We need to know exactly what is atomic.

  `x=x+1` → `r=x;r=r+1;x=r`

- The parallel composition `x=x+1 || x=3`:

  `r=x;r=r+1;x=r` || **`x=3`**

- We consider this equivalent to *any interleaving* of atomic actions (assume `x=0`, initially)

  `r=x;r=r+1;x=r;`**`x=3`**

  `r=x;r=r+1;`**`x=3`**`;x=r`

  `r=x;`**`x=3`**`;r=r+1;x=r`

  `x=3``;r=x;r=r+1;x=r`

Private vs. Shared Variables

- Actions on private variables commute with actions in other processes
- Example: assume that `r` is private and `x` is shared

```
r=x;r=r+1;x=r;x=3
r=x;r=r+1;x=3;x=r
r=x;x=3;r=r+1;x=r
x=3;r=x;r=r+1;x=r
```

Interleaving Example

Two programs update a counter (`x`)

```
P1:  x=x+1  →  r1=x;r1=r1+1;x=r1

P2:  x=x+1  →  r2=x;r2=r2+1;x=r2
```

What are the possible values of `x` after executing both P1 and P2 if `x=0` initially?

A: 0
B: 1
C: 2
D: 1 and 2
E: 0, 1, and 2

Execution Traces

When we examine execution traces, what about:

```
P1: x=0;
     while (1) {
        x=1-x;
     }
```

```
P2: y=0;
     while (1) {
        y=1-y;
     }
```

What are the possible executions that could occur?

Execution Traces

When we examine execution traces, what about:

```
P1: x=0;
    while (1) {
      while (y==0);
      x=1-x;
    }
```

```
P2: y=0;
    while (1) {
      y=1-y;
    }
```

What are the possible executions that could occur?

Mutual Exclusion

- What if two parallel processes want to access an output port?
  - Resource sharing issue
  - We'd like to be able to say:

    . . . ; <access shared resource>; . . .
  - Ensures resource is accessed by at most one process at a time

- Classic problem of *mutual exclusion*

- Commonly used to ensure a part of a program is executed atomically

Example

- Compute the sum (shared variable) of array elements in parallel by multiple processes
  - Read, update, write to ‘sum’ must be atomic

```
P1:
for(i=0;i<NUM1;i++) {
  sum += a[i];
}

P2:
for(i=NUM1;i<NUM2;i++) {
  sum += a[i];
}
```

Critical Sections

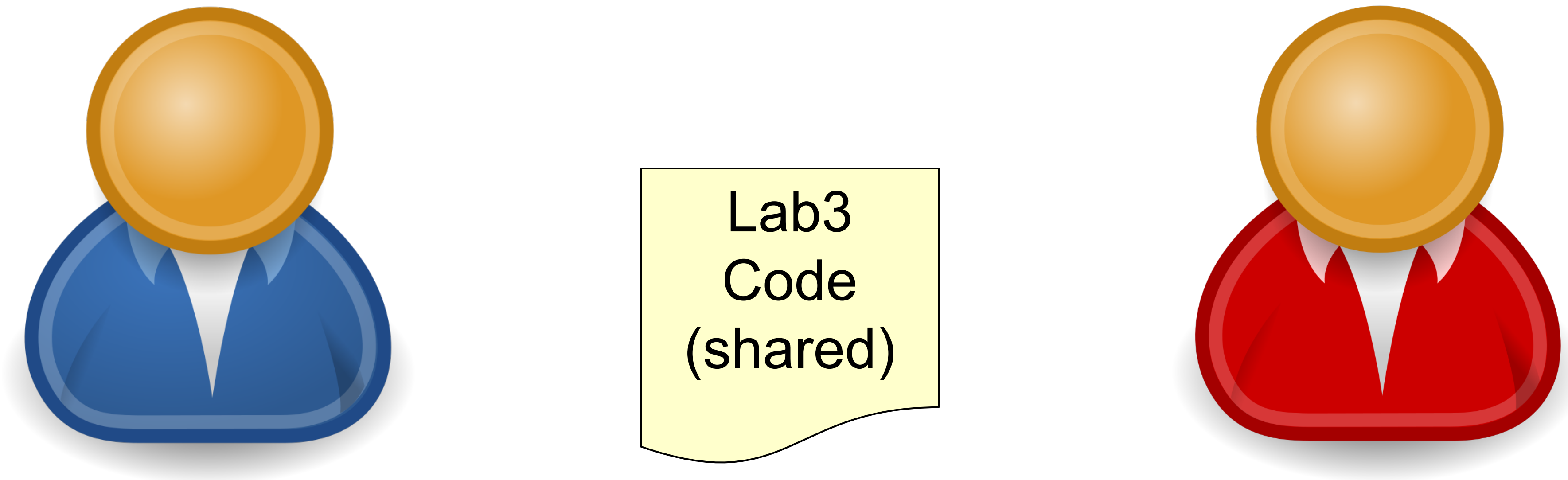
```
P1:                          P2:

   NCS1;                        NCS2;
   ...                          ...
   CS1;                         CS2;
   ...                          ...
```

- NCS: non-critical section
  - Both processes may run concurrently with arbitrary interleavings
- CS: critical section
  - Only one process should be allowed to be in a critical section

Real-Life Example: Lab Collaboration

- How to ensure that only one person edits the lab code at a time?



Mutual Exclusion: Requirements

- Safety: at any moment, at most one process is inside its CS.
- Progress: At any moment, among the processes actively contending for the CS, at least one is guaranteed access in a finite amount of time.
- Fairness: At any moment, every process actively contending for the CS is guaranteed access in a finite amount of time.

The Turn Approach

```
P1 :
while (1) {
  NCS1;
  while (turn!=1);
  CS1;
  turn = 2;
}
```

```
P2 :
while (1) {
  NCS2;
  while (turn!=2);
  CS2;
  turn = 1;
}
```

- Initially turn is either 1 or 2.
- Does this correctly implement mutual exclusion?

Dekker's Algorithm: First Attempt

```
P1 :
   NCS1;
   while (x2);
   x1=1;
   CS1;
   x1=0;
```

```
P2 :
   NCS2;
   while (x1);
   x2=1;
   CS2;
   x2=0;
```

Initially `x1 = x2 = 0`. Problem solved?

Dekker's Algorithm: Second Attempt

```
P1 :
   NCS1;
   x1=1;
   while (x2) {
      x1=0;
      while (x2);
      x1=1;
   }
   CS1;
   x1=0;
```

```
P2 :
   NCS2;
   x2=1;
   while (x1) {
      x2=0;
      while (x1);
      x2=1;
   }
   CS2;
   x2=0;
```

Dekker's Algorithm *(T. Dekker, 1966)*

```
P1 :
  NCS1;
  x1=1;
  while (x2) {
    if (turn!=1) x1=0;
    while (turn!=1);
    x1=1;
  }
  CS1;
  x1=0;turn=2;
```

```
P2 :
  NCS2;
  x2=1;
  while (x1) {
    if (turn!=2) x2=0;
    while (turn!=2);
    x2=1;
  }
  CS2;
  x2=0;turn=1;
```

Larger Atomic Actions

- If mutual exclusion is so tricky, what about more sophisticated requirements?
  - Mutual exclusion provides “larger” atomic actions
  - Perhaps we can have a mechanism to do this directly?

- There are many options:
  - Special instructions
    - Atomic test and set
    - Atomic swap
    - Atomic fetch and increment
  - Locks
  - . . .