ECE 3140 / CS 3420
EMBEDDED SYSTEMS
CONCURRENCY

Prof. José F. Martínez

TR 1:25-2:40pm in 155 Olin

Cornell University
Computer Systems Laboratory

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)

For the moment:

- Avoid the notion of physical time
- Think about how different operations are ordered

CONCURRENCY

If P_1 and P_2 are two programs (a.k.a. process):

$$P_1 \parallel P_2$$

Classify variables into two kinds:

- Shared variables: those accessed by more than one process
- Private variables: those accessed by one process

CONCURRENCY

Basic assumptions:

- Non-interference: the concurrent activities of program parts that do not share variables do not intefere 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!

CONCURRENCY

We need to know exactly what is atomic.

`x=x+1` $\triangleright$ `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:

```
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
```

Atomicity

Commutativity:

- Process 1: `x=3`
- Process 2: `x=x+1`

```
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
```

Actions on private variables *commute* with actions in other processes.

ATOMICITY

Private v/s shared variables:

$$\texttt{x=x+1} \qquad \triangleright \qquad \texttt{r=x;r=r+1;x=r}$$

$\texttt{r}$ is a private variable. Then:

$$\texttt{x=x+1} \qquad \triangleright \qquad \texttt{r=x;}\underbrace{\langle\texttt{r=r+1;x=r}\rangle}_{\text{atomic}}$$

or even:

$$\texttt{x=x+1} \qquad \triangleright \qquad \underbrace{\langle\texttt{r=x;r=r+1}\rangle}_{\text{atomic}}\texttt{;x=r}$$

Execution Traces

When we examine execution traces, what about:

P_1 :

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

P_2 :

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

What are the possible executions that could occur?

EXECUTION TRACES

When we examine execution traces, what about:

P_1 :
```
x=0;
while (1) {
  while (y == 0) ;
  x=1-x;
}
```

P_2 :
```
y=0;
while (1) {
  y=1-y;
}
```

What are the possible executions that could occur?

Weak Fairness

Assumption: weak fairness of parallel composition

- *Enabled* action: one that is ready to execute
- If an action is continuously *enabled*, it will eventually get a chance to execute

Interacting Processes

What if two parallel processes want to access an output port?

- Resource sharing issue
- We'd like to be able to say:

$$\ldots; \langle \textit{access shared resource} \rangle; \ldots$$

- Ensures resource is accessed by at most one process at a time

Classic problem of *mutual exclusion*.

Mutual Exclusion

Basic process:

P_1: `while (1) {`

$\quad NCS_1$;

$\quad \ldots$

$\quad CS_1$;

$\quad \ldots$

`}`

P_2: `while (1) {`

$\quad NCS_2$;

$\quad \ldots$

$\quad CS_2$;

$\quad \ldots$

`}`

NCS: non-critical section; need not terminate
CS: critical section; always terminates

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

P_1 :
```
while (1) {
    NCS1;
    while (turn!=1) ;
    CS1;
    turn = 2;
}
```

P_2 :
```
while (1) {
    NCS2;
    while (turn!=2) ;
    CS2;
    turn = 1;
}
```

Initially `turn` is either 1 or 2.

Does this solve the problem?

Dekker's Algorithm: First version

P_1:
```
while (1) {
    NCS1;
    while (x2);
    x1=1;
    CS1;
    x1=0;
}
```

P_2:
```
while (1) {
    NCS2;
    while (x1);
    x2=1;
    CS2;
    x2=0;
}
```

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

DEKKER'S ALGORITHM: SECOND VERSION

P_1 :
```
while (1) {
    NCS1;
    x1=1;
    while (x2) {
      x1=0;
      while (x2);
      x1=1;
    }
    CS1;
    x1=0;
}
```

already set intent; give up
if show intent
But, both are giving up =>
after you after you livelock

Text

P_2 :
```
while (1) {
    NCS2;
    x2=1;
    while (x1) {
      x2=0;
      while (x1);
      x2=1;
    }
    CS2;
    x2=0;
}
```

Yields: GB algorithm; super polite
(x1,x2) express intent; always
checking
Safe: yes
Progress: livelock; NOT safe
Both express intent at the same;
both will yield each other
“After-you after-you livelock
problem”

Problem solved?

DEKKER'S ALGORITHM

EXAM:
P1 not interested, can P2 access P2
If both locked, does at least one get into CS
if both are accessing lock step, do both end up in CS
If one is already in CS can the other go into the CS?
LOOK AT ONLINE LECTURE 3/3

expresses intent

while x1 is also interested

if it is not P2's turn, it will reli
give up ONLY if it's not my tur

*If both show intent, turn is the tie breaker

finally my turn

P_1 :
```
while (1) {
    NCS1;
    x1=1;
    while (x2) {
      if (turn!=1) x1=0;
      while (turn!=1);
      x1=1;
    }
    CS1;
    x1=0;turn=2;
}
```

P_2 :
```
while (1) {
    NCS2;
    x2=1;
    while (x1) {
      if (turn!=2) x2=0;
      while (turn!=2);
      x2=1;
    }
    CS2;
    x2=0;turn=1;
}
```

If P1 is in CS, turn = 2; then NEVER give up (x2 to avoid after-you after-you)
second while loop turn check so that x2 doesn't give up; in P1 it will yield to P2

This ensures fairness (by not giving up intent if it's a processes turn). Even if x1 finishes CS and goes back into NCS1, P2 will eventually run and get into CS as well

(Due to J. Dekker, published by E.W. Dijkstra in 1968)

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
- ...

Locks

A lock `l` supports two basic operations:

- `lock(l)` (sometimes called *acquiring* a lock)
- `unlock(l)` (sometimes called *releasing* a lock)

P_1 :
```
while (1) {
    NCS1;
    lock(l);
    CS1;
    unlock(l);
}
```

P_2 :
```
while (1) {
    NCS2;
    lock(l);
    CS2;
    unlock(l);
}
```