ECE3140 / CS3420
Embedded Systems

Locks

Prof. José F. Martínez
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;

Weaknesses?
Outline

- Lock: a synchronization primitive to efficiently support mutual exclusion
- Definition and usage example
- Implementation
  - Atomic read-modify-write instructions
  - Spinlocks
  - Blocking locks
- Building higher-level constructions using locks
New Abstraction: Locks

- A lock $l$ supports two basic operations:
  - \texttt{lock}(l) (sometimes called acquiring a lock)
  - \texttt{unlock}(l) (sometimes called releasing a lock)

\[
\begin{align*}
\text{P1} & : & \text{NCS1;} & \text{lock}(l); & \text{CS1;} & \text{unlock}(l); \\
\text{P2} & : & \text{NCS2;} & \text{lock}(l); & \text{CS2;} & \text{unlock}(l); \\
\end{align*}
\]
Why Use a Variable (‘l’)?

- What if there are multiple resources that need to be protected with a lock?

P1:

lock( );
x=x+1;
unlock( );

P2:

lock( );
x=x+1;
unlock( );

Note: the lock variable (l) is NOT a variable that the lock is protecting!
Deadlock

- Consider nested locks

P1

: lock(a);
: lock(b);
: CS1;
: unlock(b);
: unlock(a);
: 

P2

: lock(b);
: lock(a);
: CS2;
: unlock(a);
: unlock(b);
: 

Atomicity through Disabling Interrupts

- Timer interrupts are used to switch between processes
  - To avoid that, disable interrupts!

- On a uni-processor system, small atomic actions can be performed by disabling interrupts
  - No interrupt within a critical section

- Not a good solution in general
Broken Mutual Exclusion Algorithm

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

P2:
   NCS2;
   while (x1);
   x2=1;
   CS2;
   x2=0;
Atomic Read-Modify-Write Instruction

- Mutual exclusion can be implemented using ordinary load and store instructions
  - However, protocols for mutual exclusion are difficult to design...

- Simpler solution:
  - Atomic read-modify-write instructions

Examples: *m is a memory location, R is a register*

- Test&Set (m), R:
  - R ← M[m];
  - if R==0 then
    - M[m] ← 1;

- Fetch&Add (m), Rv, R:
  - R ← M[m];
  - M[m] ← R + Rv;

- Swap (m), R:
  - Rt ← M[m];
  - M[m] ← R;
  - R ← Rt;
Blocking Locks

- Avoid unnecessary spinning
  - If another process owns a lock, suspend a process
    - Maintain a list of blocked processes for each lock
  - Wake up a waiting process when a lock is released
Higher-Level Constructs

Locks can be used to build higher-level constructs

Example: Readers and Writers

- Two types of processes
  - Reader: reads a shared resource
  - Writer: modifies a shared resource

- Safety goals:
  - Reads and writes are mutually exclusive
  - Writes are mutually exclusive

- Provide:
  - $\text{enter}_r$, $\text{exit}_r$
  - $\text{enter}_w$, $\text{exit}_w$
Approach

- A simple approach: two shared variables
  - \( nw \): number of writers
  - \( nr \): number of readers
enter_r:
    lock(m);
    while (nw) {
        unlock(m);
        while (nw);
        lock(m);
    }
    nr=nr+1;
    unlock(m);

enter_w:
    lock(m);
    while (nw>0 || nr>0) {
        unlock(m);
        while (nw>0 || nr>0);
        lock(m);
    }
    nw=1;
    unlock(m);