Synchronization

Critical sections

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Concurrency and Synchronization

- Development environment
  - concurrent programming
  - cooperating processes or threads
- Issues
  - Need to manipulate shared data
  - Race conditions may arise
  - There may be sections of not-reentrant code
- Solution strategy
  - appropriately synchronize the cooperating processes, to make their results not dependent on their relative speed

Results dependent on the order of execution

Uninterruptible code

"Too much milk problem"

Time	Person A	Person B
10.00	Look in fridge; out of milk	
10.05	Leave for store	
10.10	Get to the store	Look in fridge; out of milk
10.15	Buy milk	Leave for store
10.20	Back home	Get to the store
10.25	Put milk in fridge	Buy milk
10.30		Back home
10.35		Put milk in fridge Hops!!!

LIFO - Stack

P_i / T_i

```
void push (int val) {
  if(top>=SIZE-1)
    return;
  top++;
  stack[top] = val;
  return;
}
```

P_j / T_j

```
int pop (int *val) {
  if(top<=-1)
    return;
  *val=stack[top];
  top--;
  return;
}
```

- **push** and **pop**
  - Operate on the same end of the stack
  - Variable **top** is shared

top++ then top-- or viceversa Problems?!

Can overwrite a value (lose a push), make a pop of a nonexistent value, etc.

FIFO – Queue – Circular Buffer

P_i / T_i

```
void enqueue (int val) {
  if (n>SIZE) return;
  queue[tail] = val;
  tail=(tail+1)%SIZE;
  n++;
  return;
}
```

register = n
register = register + 1
n = register

P_j / T_j

```
int dequeue (int *val) {
  if (n<=0) return;
  *val=queue[head];
  head=(head+1)%SIZE;
  n--;
  return;
}
```

register = n
register = register - 1
n = register

- **enqueue** and **dequeue**
  - Operate on the different ends of the queue, using two variables **tail** and **head**
  - Variable **n** is still shared

Increments and decrements can be lost

Race prevention

- The race conditions could be prevented if
  - No tread executes these functions simultaneously
  - No other tread can execute a function when another tread is running the other function
  - The code of the two functions is executed in **mutual exclusion** to fulfill Bernstein's conditions

Critical sections

- Critical Section (CS) or Critical Region (CR)
  - A section of code, common to multiple threads, in which they can access (read and write) shared objects
  - A section of code in which multiple threads are competing for the use (read and write) of shared resources (e.g., data or devices)
  - Problem – ensure that when a thread is executing in its CS, no other thread is allowed to execute in its CS (same or different code).
  - Solution – establish an access protocol to enter the critical section in **mutual exclusion**.

Solution

- Establish an **access protocol** that enforces **mutual exclusion** for each CS
  - A thread executes a ”reservation” code before entering a CS
  - The reservation code blocks (locks out) the thread if another thread is using its CS
  - Leaving its CS, a thread executes a code to release the CS
  - The release possibly unlocks another thread which was waiting in the ”reservation” code of its CS

Access protocol

P_i / T_i

```
while (TRUE) {
  ...
  reservation code
  Critical Section
  release code
  ...
  non critical section
}
```

P_j / T_j

```
while (TRUE) {
  ...
  reservation code
  Critical Section
  release code
  ...
  non critical section
}
```

- Every CS is protected by an
  - enter code (reservation, or prologue)
  - exit code (release, or epilogue)

Solutions

- Software functions
  - Solutions without special CPU instructions
- Hardware
  - Solutions based on special hardware characteristics, or special CPU instructions
- System calls
  - The kernel provides the data structures, and the related system calls, that the programmer can properly use for solving the mutual exclusion problem

Semaphore: introduced by Dijkstra [1965]