# Real-Time Embedded Systems

DT8025, Fall 2016
http://goo.gl/AZfc9l

Lecture 4

Masoumeh Taromirad
m.taromirad@hh.se

HÖGSKOLAN I HALMSTAD

Center for Research on Embedded Systems
School of Information Technology

# Critical Section Problem

Revisited!

## Critical Section

- part of a multi-threaded program that may not be concurrently executed by more than one of the program's processes.
- typically, accesses a shared resource.

# Critical Section Problem

## Solution properties

### Mutual Exclusion

One thread of execution never enters its critical section at the same time that another, concurrent thread of execution enters its own critical section.

# Critical Section Problem

## Solution properties

### Progress

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

# Critical Section Problem

## Solution properties

### Bounded Waiting (starvation-free, finite bypass)

There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted.

Starvation happens when a process is perpetually denied necessary resources to process its work.

# Mutual Exclusion: An example algorithm

```
flag[0]: false, flag[1]: false, turn: 0 or 1
```

p0:

```
flag[0] = true
while (flag[1]) {
    if (turn = 1) {
        flag[0] = false
        while (turn = 1) {}
        flag[0] = true
    }
}
// critical section
turn = 1
flag[0] = false
// remainder section
```

p1:

```
flag[1] = true
while (flag[0]) {
    if (turn = 0) {
        flag[1] = false
        while (turn = 0) {}
        flag[1] = true
    }
}
// critical section
turn = 0
flag[1] = false
// remainder section
```

# Context Switching

The process of storing and restoring the state (more specifically, the execution context) of a process or thread so that execution can be resumed from the same point at a later time.

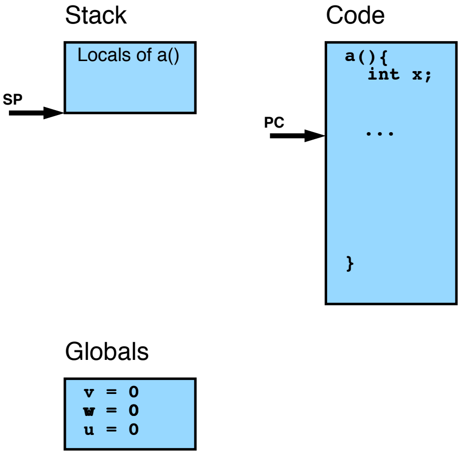
## Stack Pointer

A small register that stores the address of the last program request in a stack.

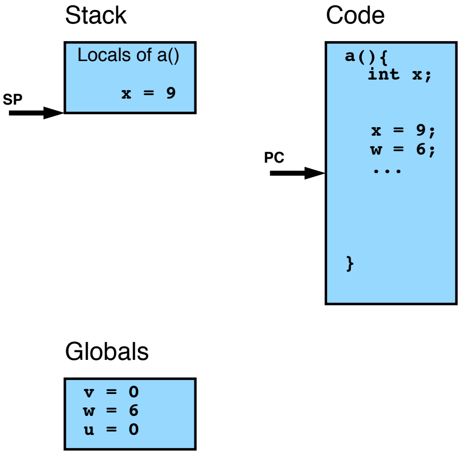
## Program Counter

A processor register that indicates where a computer is in its program sequence.

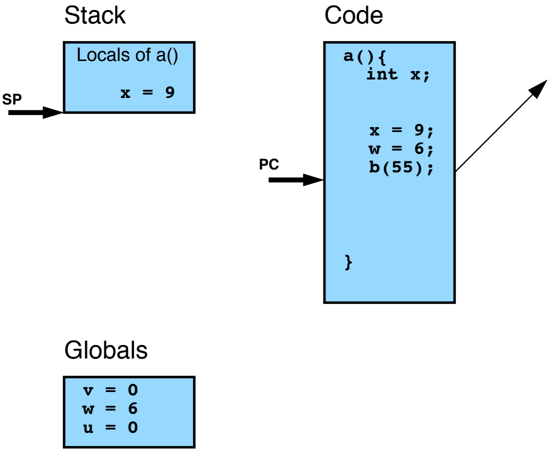
# Execution of a C program



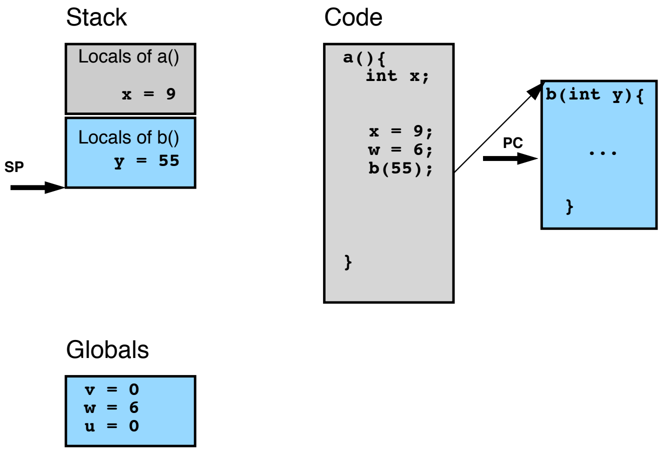
# Execution of a C program



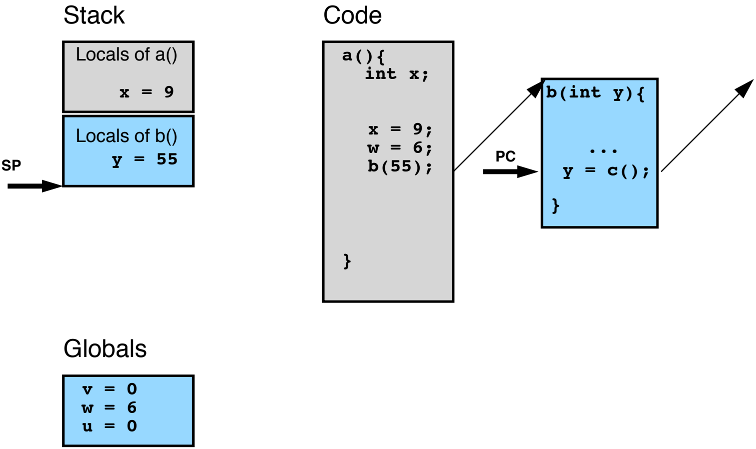
# Execution of a C program



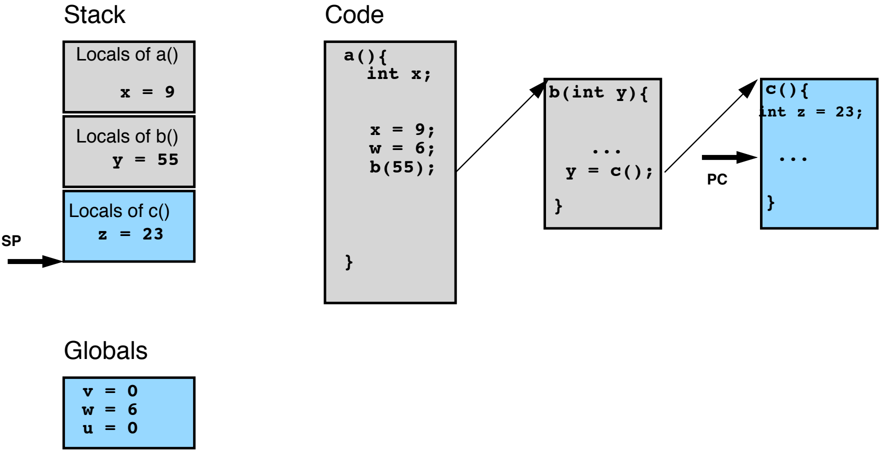
# Execution of a C program



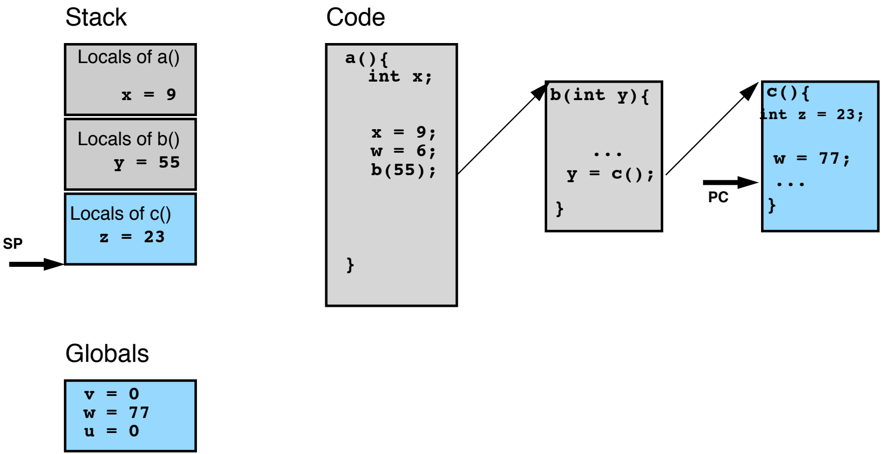
# Execution of a C program



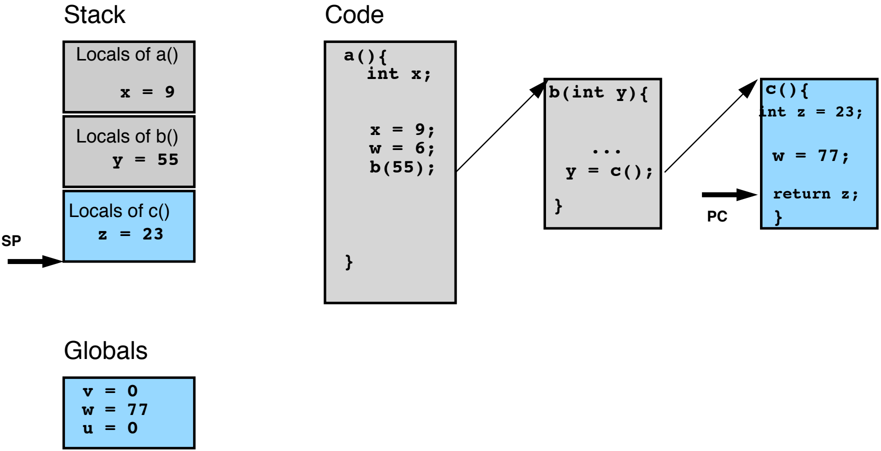
# Execution of a C program



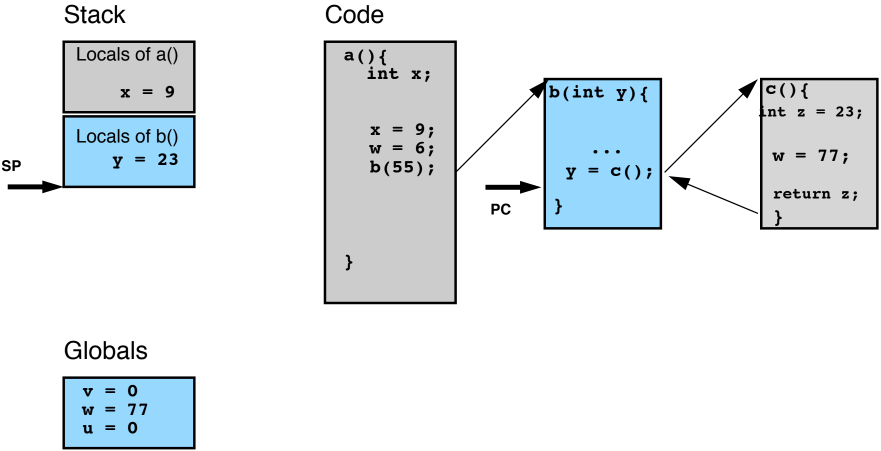
# Execution of a C program



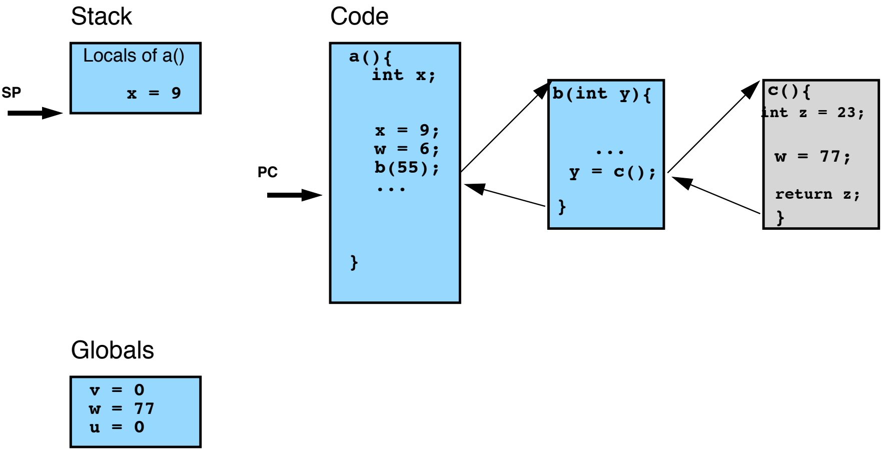
# Execution of a C program



# Execution of a C program



# Execution of a C program



# Concurrent Programs?

Imagine we had 2 CPUs, then we could run two programs at the same time!

One way of programming this in only 1 CPU is to keep track of 2 stack pointers and 2 program counters!

# What is it about?

```
struct Params params;
```

```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}
```

We want to provide means for these two mains to execute concurrently! As if we had 2 CPUs!

# What might a program look like?

```
main(){
  create_thread(decoder_main);
  controller_main();
}
```

Notice that the function `create_thread` takes a *function* as an argument!

The role of `create_thread` is to provide one extra **Program Counter** and **Stack Pointer**.

# What we need ...

We will have to keep track of the threads, so we introduce a data structure describing a thread.

```
struct Thread_Block{
   void    (*fun)(int)   // function to run
   int     arg;          // argument to the above
   ucontext context;      // pc and sp
   ...                   // ...
};
typedef struct Thread_Block *Thread
```

We will keep track of threads using global variables for

1. a queue of `Threads`: the ready queue
2. and the `current` thread.