Chapter 4_1

Operating Systems and Concurrent Programs

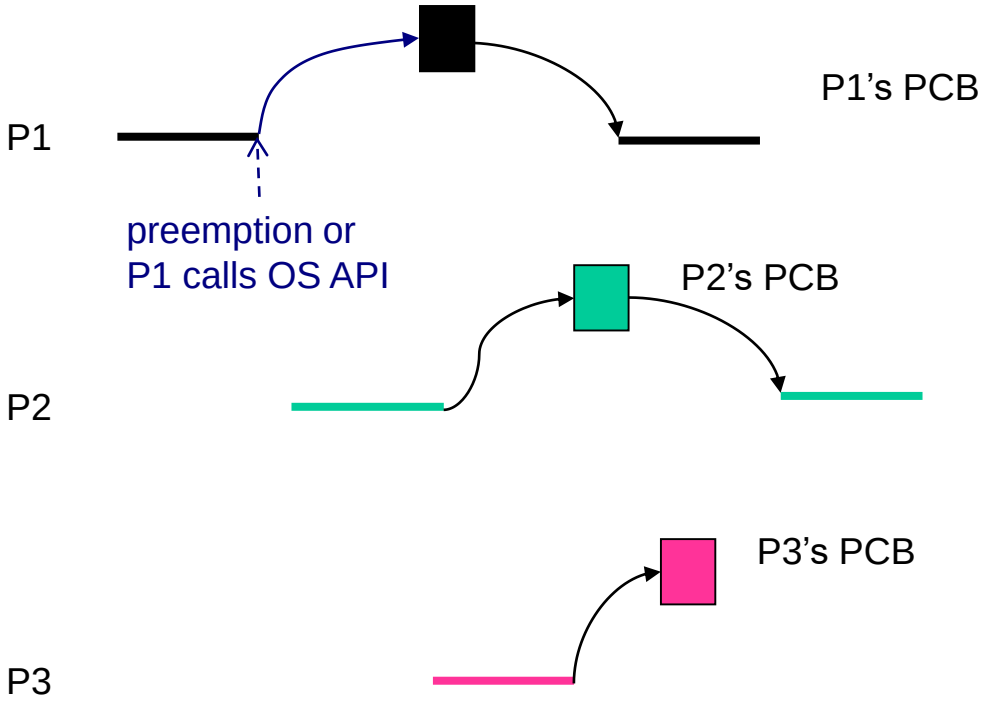
Concurrent Programs and BrickOS internals

- Processes and threads
- Synchronization
- BrickOS Internals

Processes

- All the runnable software on a computer, including the OS, is organized into a number of sequential processes (or processes in short)
- In a multiprogramming system, the CPU switches from program to program, giving the users the illusion of parallelism (pseudo parallelism)

Processes (cont)



Process switching on a single CPU

Processes (cont)

- Process:
  - it is a sequentially executing instance of a program (old definition),
  - from the OS's point of view, it is a scheduling unit

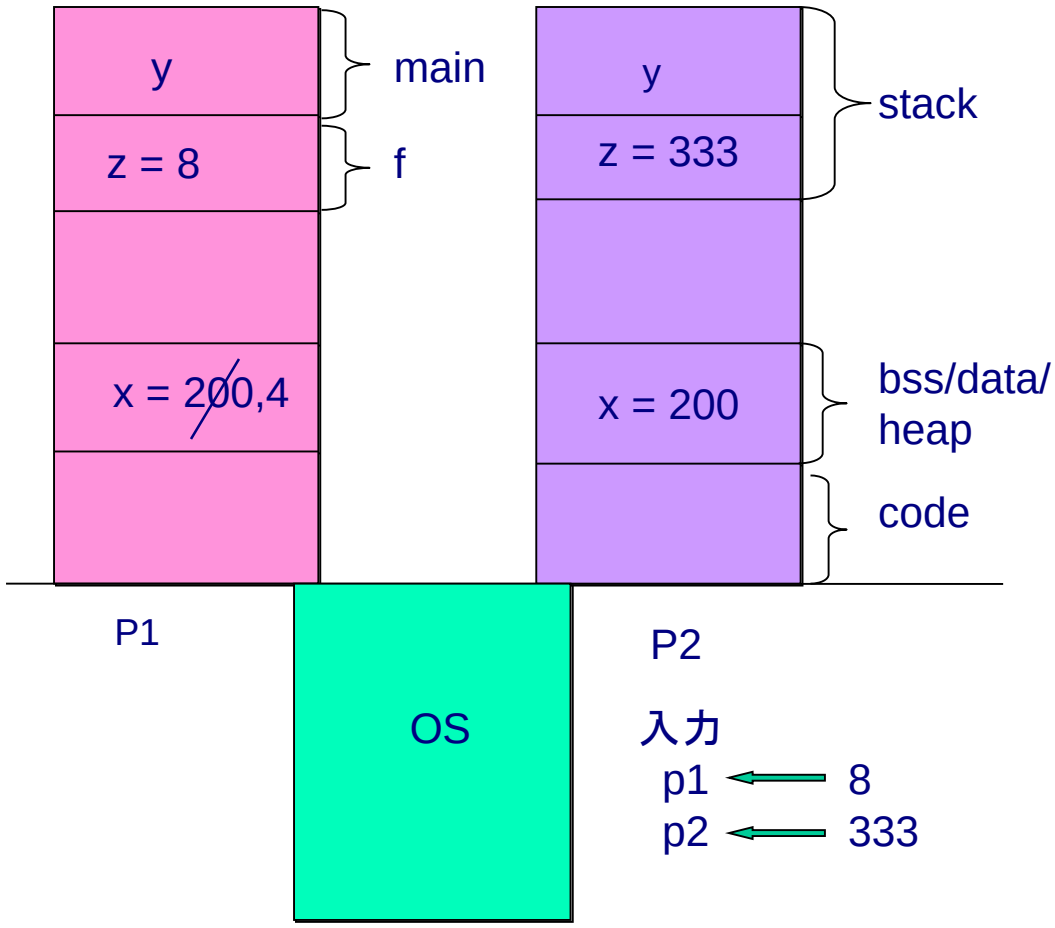
- For each process, OS maintains the following
  1. program's executable code, data, and stack (and the memory areas)
  2. copies of program counter, stack pointer, and other CPU registers
  3. all other information necessary to run the program: process id, priority, accounting information, memory information, open files, process state, etc.

  Note: 2 and 3 are stored in OS data structures called *process control blocks (PCBs)* --- an array (or linked list) of structures, one for each process currently exists.

- Clearly understand the *difference between program and process*
  - Example： three users execute Emacs at the same time; three processes execute the Emacs program

Processes (cont)

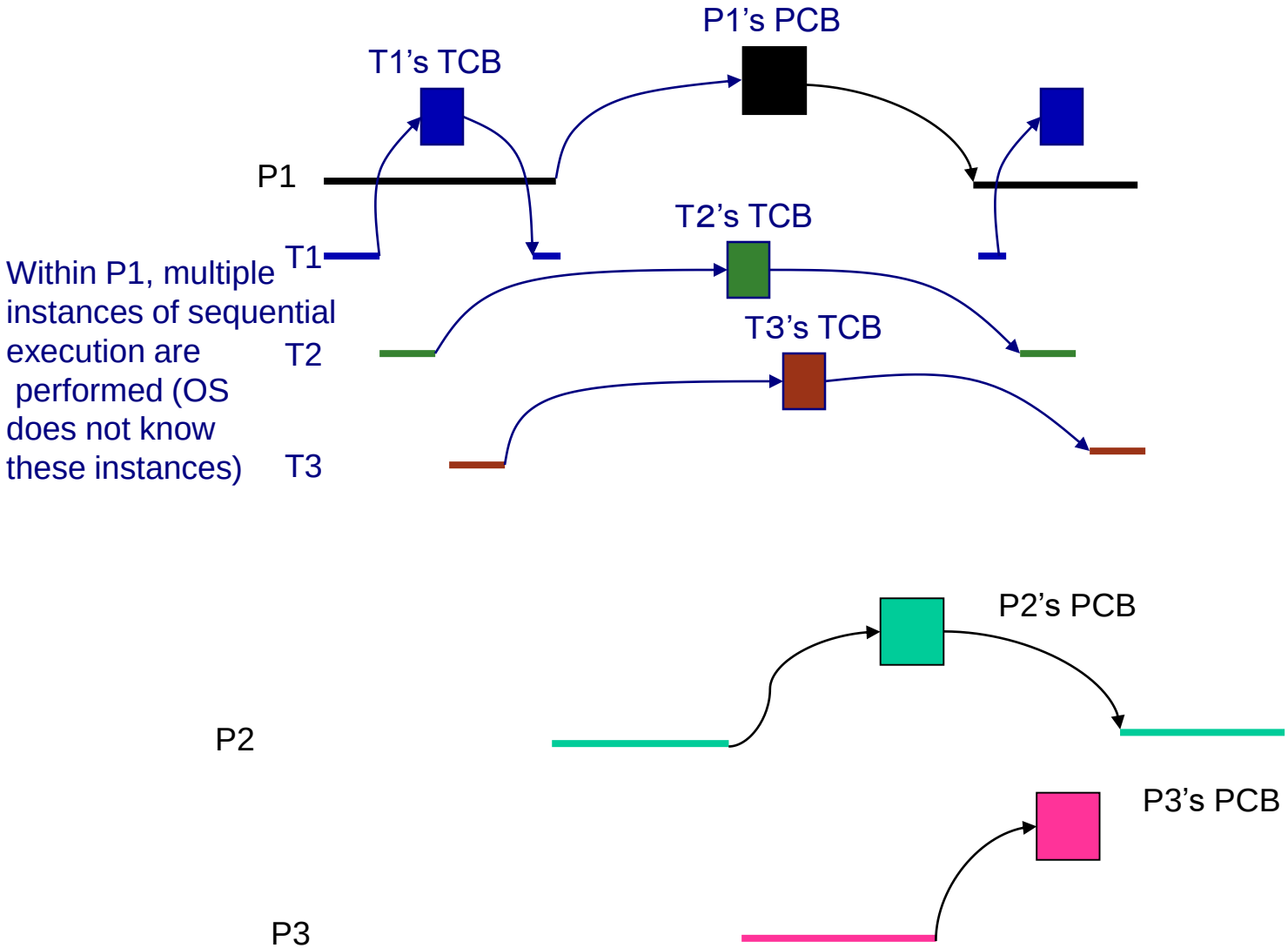
```
int x = 200;
main( ) {
   int y;
   f( );
}
f( ) {
   int z;
   z = read();
   x = 4;
   print x;
}
```



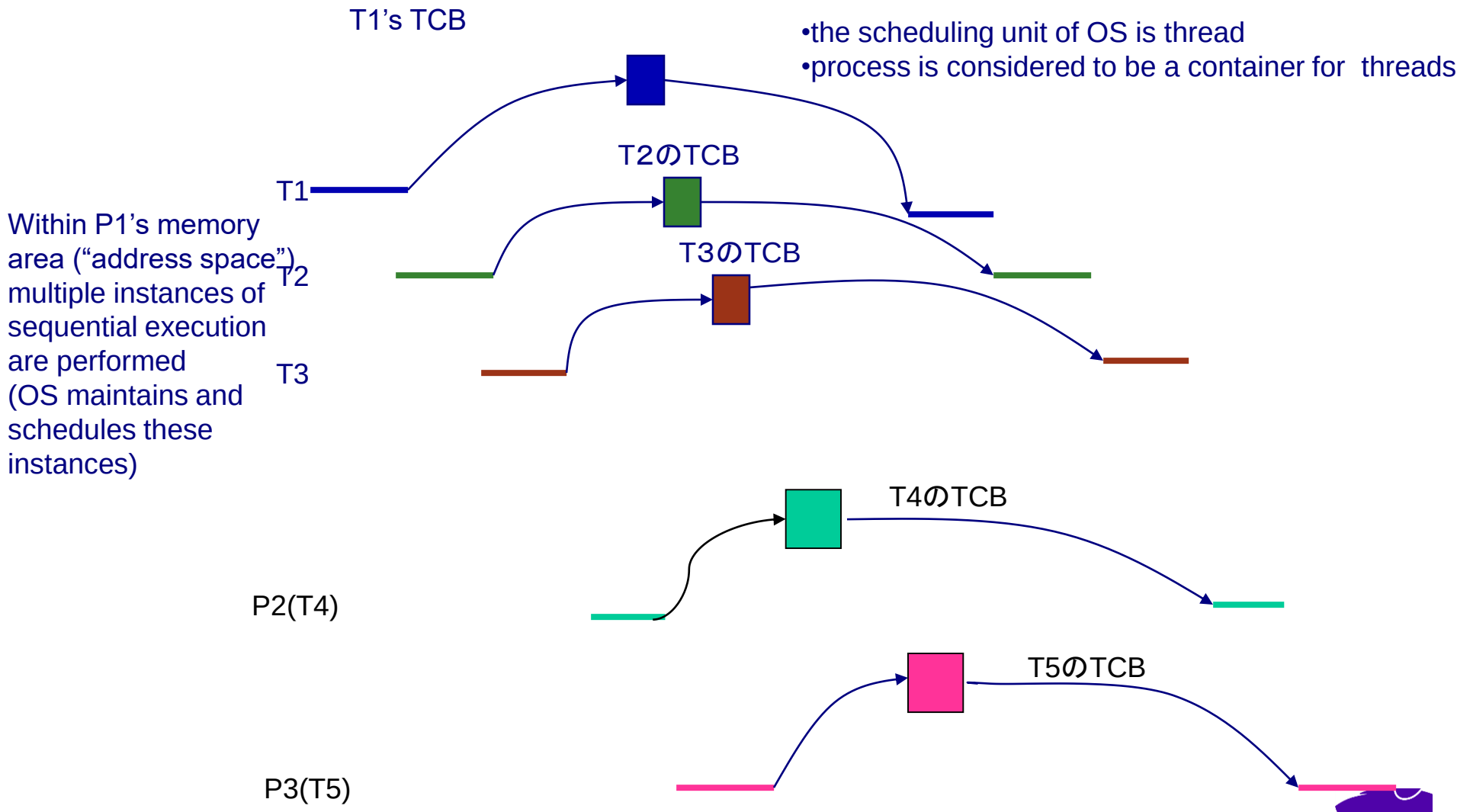
Preemption and Priority Scheduling

- There may be more than one process in the system
- OS chooses one process at a time and give the CPU to it ("Scheduling")
  - OS maintains a queue of processes ("Ready Queue") and gives the CPU to the process at the head of the queue
- A process under execution returns the CPU to the OS in the following ways
  - A process requests I/O operations, such as accessing Console, Disk, Network. If the operation does not complete immediately, OS "blocks" this process and schedules another process
  - A process executes synchronization operations (primitives) to block itself or a "yield" operation to return the CPU voluntarily
  - OS predetermines the maximum time to allocate the CPU to each process (called "time slice" or "time quantum"). If a process has run the time slice, OS takes up the CPU (this is called "preemption")
    - there are OSs which do not do "preemption"
- Some OSs manages "priorities" among processes and scheduling processes based on their priorities (priority scheduling)

Threads (Implementation outside OS)



Threads (Implementation by OS)



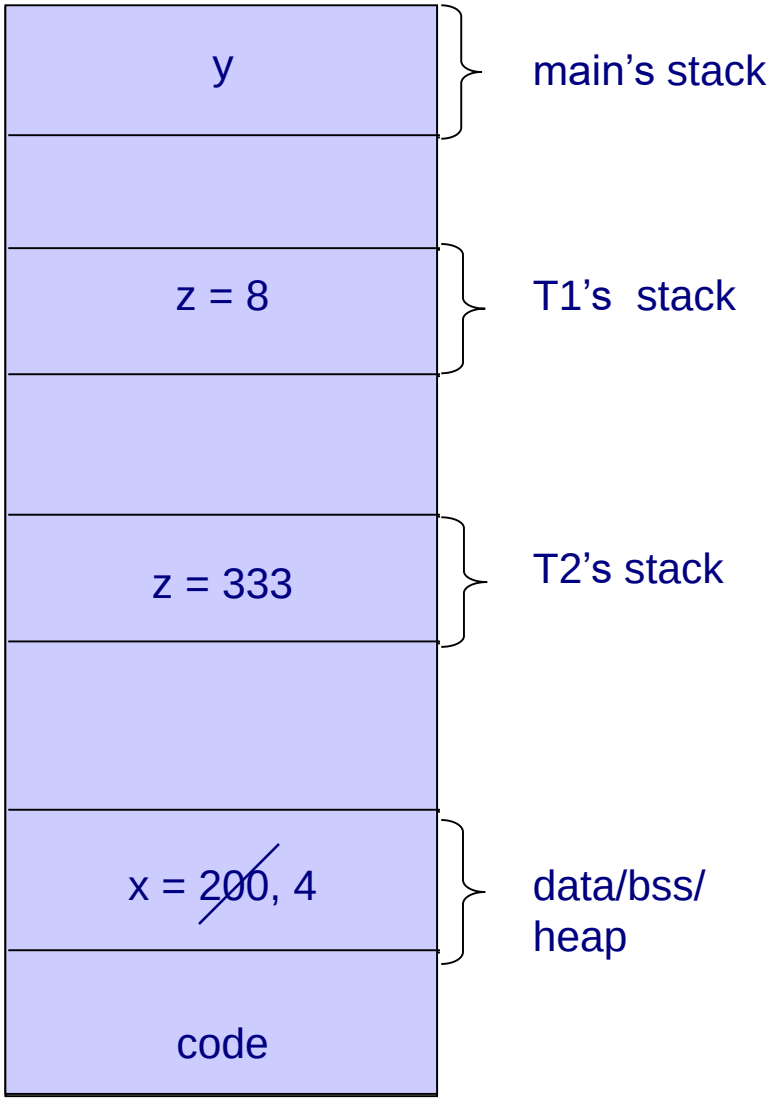
Thread Switching

```
int x = 200
main( ) {
   int y;
   create_thread(f); // T1
   create_thread(f); // T2
   go_thread( );
}
f( ) {
   int z;
   z = read( );
   x = 4;
   print x;
}
```

T2

T1

Input : 8 333



Why We Want Concurrent Programs

- To increase CPU utilization (general purpose systems)
- To provide fair service to many clients (servers including time-sharing systems)
- To improve response time in GUI based applications (by assigning blocking jobs to background worker threads)
- To allow programmers to focus only on sequential execution of each thread
  - Actually, synchronization code is required to coordinate the execution of the multiple threads (difficult to develop)
- To allow priority control among various execution entities (hard real time systems)
  - Hard real-time systems want to reduce the number of threads (to minimize the overhead). The reason to use multiple threads in such systems is to do priority control (e.g., Rate-Monotonic based Periodic-Task system)
- To allow information to be shared among multiple threads for collaborattion (variables and objects in data/bss/heap)
  - We need to understand that it is also dangerous to share data among threads (mutual exclusion)

Concurrent Programs and Synchronization

Processes and Thread

Synchronization

BrickOS Internals

Why do we need synchronization ?

- Mr. and Mrs. Mizuno own a bank account B
- Currently B's balance is $100
- Mr. Mizuno is about to deposit $50 in account B at ATM1
- Mrs. Mizuno is about to deposit $150 in account B at ATM2
- All ATMs of this bank execute the following code

`deposit` program

```
deposit (account, amount)
   local var x;
   begin
      x := account.getBalance();
      x := x +  amount;
      account.setBalance(x);
   end
```


Execution Trace

Mr. Mizuno :

```
x_1 := B; --- (1)
x_1 := x_1 + 50; --- (2)
B := x_1; --- (3)
```

Mrs. Mizuno:

```
x_2 := B; --- (4)
x_2 := x_2 + 150; --- (5)
B := x_2; --- (6)
```

Suppose thread switching takes place as follows:
(1) (4) (5) (6) (2) (3) :

```
x_1 := 100;  --- (1)    /* x_1 == 100 */
x_2 := 100;  --- (4)    /* x_2 == 100 */
x_2 := x_2 + 150; --- (5)    /* x_2 == 250 */
B := x_2;  --- (6)    /* B == 250 */
x_1 := x_1 + 50;  --- (2)    /* x_1 = 150 */
B := x_1;          --- (3)    /* B == 150 */
```


Race Condition and Mutual Exclusion

- Race Condition：
  - Two or more processes read or write some shared data and the final result depends on who runs precisely when. The way to avoid race conditions is to guarantee mutual exclusion
- Mutual Exclusion：
  - If one process is accessing a shared variable, the other processes will be excluded from accessing the same variable
- Critical Section：
  - segment of a program that accesses the shared memory, and therefore has to be treated as an “indivisible part”
- In the ATM program, the three lines in function deposit( ) constitutes a critical section
  - The race condition is avoided by executing these three lines indivisibly
- In mutual exclusion, the most difficult part is to identify what parts of code constitute critical sections – not what parts access shared data
  - you will experience the difficulty in the implementation of TippySr

Condition Synchronization

- There is another type of synchronization, called “condition synchronization”
- Condition synchronization delays execution of a thread until some event arrives or the system enters a certain state
- Examples of condition synchronization include:
  - the producer/consumer problem: producer threads put messages into a buffer and consumer threads get the messages from the buffer
    - a producer thread is delayed to put a message until a consumer thread has picked up the previous message and emptied the buffer
    - a consumer thread is delayed to get a message until a producer thread has put the next message
  - the barrier synchronization: two threads wait for each other until both arrive at their respective program points
  - the readers/writers synchronization: multiple threads access a buffer by satisfying the following requirements：
    - a writer thread can access the buffer only if no other writer thread or no reader thread is accessing the buffer
    - a reader thread can access the buffer only if no writer thread is accessing the buffer

Synchronization Exercises

- To understand difficulties involved in concurrent programs and to deepen the understanding of synchronization, we see traditional synchronization exercise found in many basic OS text books
- We try to develop synchronization code to implement mutual exclusion
  - Actual synchronization primitives (the most fundamental service functionalities that Operating Systems provide) are not implemented in the way we discuss here
  - We will see how real Operating Systems implement synchronization primitives later
  - We assume that there are only two processes (threads)
- Synchronization primitives must satisfy the following four conditions:
  1. Two threads never enter their critical sections (CSs) at the same time (mutual exclusion).
  2. If one thread arrives at the entry point of its CS while the other thread has not arrived at the entry point of its CS, the former thread must enter its CS immediately.
  3. If a thread arrives at the entry point of its CS, it must enter its CS in finite time.
  4. (this condition applies to all of the above) A thread preemption may occur at any time, and in a multi-processor system, you cannot assume anything about the relative speeds of the processors.

Synchronization Exercises (cont)

- We consider the following code structure
  - Develop code for "**entry section**" and "**exit section**"

```
T1:   repeat
                preceding section
              entry section
                critical section
              exit section
                remainder section
            until false

 T2:   repeat
                preceding section
              entry section
                critical section
              exit section
                remainder section
            until false
```

Synchronization Exercises (cont)

- Try 1

```
var turn:   1..2;

 T1: repeat
       while turn != 1 do skip;
            critical section
       turn := 2;
            remainder section
     until false

 T2: repeat
       while turn != 2 do skip;
            critical section
       turn := 1;
            remainder section
     until false
```

Synchronization Exercises (cont)

- Try 2

```
var  flag : array [1..2] of boolean;
     flag[1] := flag[2] := false;
     /* flag[i] is true iff P_i is executing in its
        critical section. */

 T1: repeat
       while flag[2] do skip;
       flag[1] := true;
               critical section
       flag[1] := false;
               remainder section
     until false;

 T2: repeat
       while flag[1] do skip;
       flag[2] := true;
               critical section
       flag[2] := false;
               remainder section
     until false;
```

Synchronization Exercises (cont)

- Try 3

```
var  flag : array [1..2] of boolean;
     flag[1] := flag[2] := false;
       /*  flag[i] is true if Pi wants to enter
           the critical section . */

  T1: repeat
        flag[1] := true;
        while flag[2] do skip;
                critical section
        flag[1] := false;
                remainder section
      until false;

  T2: repeat
        flag[2] := true;
        while flag[1] do skip;
                critical section
        flag[2] := false;
                remainder section
      until false;
```

Synchronization Exercises (cont)

- Try 4

```
var  flag : array [1..2] of boolean;
     flag[1] := flag[2] := false;

  T1: repeat
   L1:  while flag[2] do skip;
        flag[1] := true;
        if flag[2] then {flag[1] := false; goto L1;}
                critical section
        flag[1] := false;
                remainder section
      until false;

  T2: repeat
   L2:  while flag[1] do skip;
        flag[2] := true;
        if flag[1] then {flag[2] := false; goto L2;}
                critical section
        flag[2] := false;
                remainder section
      until false;
```

Peterson's Algorithm（1981）

```
var  flag : array [1..2] of boolean;
     flag[1] := flag[2] := false;
     turn : 1..2;
```

```
T1: repeat
      flag[1] := true;
      turn := 2;
      while (flag[2] and turn == 2) do skip;
         critical section
      flag[1] := false;
         remainder section
    until false;
```

```
T2: repeat
      flag[2] := true;
      turn := 1;
      while (flag[1]  and turn == 1) do skip;
          critical section
      flag[2] := false;
          remainder section
    until false;
```


Correctness Argument of Peterson's Algorithm

1. Mutual Exclusion:
   - Assume that threads T1 and T2 are in their critical sections at the same time. Then, there are only the following two cases:
   - Case 1： One thread (say T1) is already in its CS、and another thread (say T2) arrives at the entry point of its CS (its while statement)
     - Then, flag[1]==true and Turn==1 hold, and this condition never changes until T1 leaves its CS.
     - Therefore, this case never occurs.
   - Case 2： Both threads (T1 and T2) arrive at their while statements at the same time.
     - In this case, flag[1]==true and flag[2]==true hold (otherwise, it becomes Cas 1)
     - For T1 to be in its CS, turn==1 must have held when it arrived at its while statement. This mean that T1 had executed "turn:=2;" before T2 executed "turn:=1;"
     - Similarly, for T2 to be in its CS, T2turn==2 must have held when it arrived at its while statement. This means that T2 had executed "turn:=1;" before T1 executed "turn:=2;"
     - Contradiction

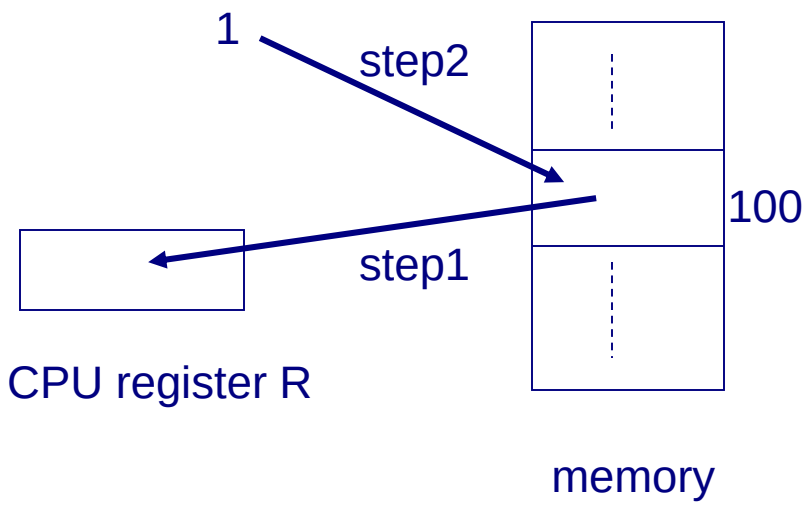
Correctness Argument of Peterson's Algorithm (cont)

2. If one thread (T1) arrives at the entry point of its CS while the other thread (T2) has not arrived at the entry point of its CS, T1 must enter its CS immediately.
   - In this case, since flag[2]==false holds, T1 can enter its CS

3. If one thread (T1) arrives at the entry point of its CS, T1 must enter its CR in finite time
   - As shown in the above 2, if flag[2]==false, T1 can enter its CS immediately.
   - Now consider a case in which flag[2]==true holds
     - If turn==1 holds, T1 can enter its CS immediately
     - If turn==2 holds (T2 had executed turn:=1 before T1 executes turn:=2), T2 is in its CS. When T2 exits its CS, it sets flag[2] to false.
       - If T1 is scheduled and finds flag[2]==false, T1 enters its CS.
       - If T2 is scheduled and came back to the entry of its CS again, T2 sets flag[2] to true. However, T2 sets turn to 1 and therefore, T2 cannot enter its CS and T1 enters its CS as soon as it is scheduled.
   - As seen above, T1 can enter its CS if it waits for T2 to execute its CS at most once.

Hardware Support for Mutual Exclusion

- Disabling Interrupt
  - Any problem ?
- Test and Set (TS) Instruction
  - Any problem ?



Initial value in memory
at address100 is 0

```
L1:  TS   R, 100
     JNZ  R, L1
```

critical section

```
LD  100, #0
```


Semaphore

- “Disabling Interrupt” is very dangerous (it may shut down the system) and cannot open up to general users
- Mutual Exclusion based on “TS instruction” is called “busy waiting”
  - if the number of processors is less than the number of processes, it is inefficient
  - if a thread cannot enter its critical section, it waits by looping the code (wasting the CPU)
- Dijkstra proposed a new data type called a "semaphore” to implement synchronization
  - If a thread cannot enter its critical section, the semaphore “blocks” the thread
    - remove the thread from the Ready Queue in order not to allocate the CPU

Semaphore

- Semaphore type provides two operations: P( ) and V( )
- The implementation of Semaphore is as follows:

```
class Semaphore {
    private:
      count: non-negative integer;
      Q:  queue of TCB;

    public:
      Semaphore(int x) {count := x;}
      void P( );
      void V( );
}
```

Semaphore（cont）

- Implementation of P( ) and V( ) is

```
void P( ) {
  if (count > 0) count := count - 1;
  else {
       add the calling thread to Q;
       block the calling thread (remove the thread from the ready queue);
    }
}


void V( ) {
   if (one or more threads are waiting on Q) {
      remove a thread T from Q;
      wake up T (place T in the ready queue);
   } else  count := count + 1;
}
```

Note that P() and V() operations themselves are critical sections. How is mutual exclusion on these operations enforced ?

Semaphore（cont）

- The meaning of Semaphore operations
  - the initial value of count is positive (count> 0)： resource management
    - count : represents the number of resource items to manage
    - P()：consumes (borrows) one resource item
    - V()：produces (returns) one resource item
  - the initial value of count is 0 (count = 0) ：blocking the calling process
    - P()：blocks itself
    - V()：releases one process
      - If there is no blocked process in the semaphore, the semaphore remembers this state and the next process that issues P() will not be blocked

Mutual Exclusion using Semaphore

- Implementation of mutual exclusion using Semaphore

```
var Semaphore s = new Semaphore(1);
```

```
T1: repeat
      s.P( );
         critical section
      s.V( );
         Remainder Section
    until false;
```

```
Tn: repeat
      s.P( );
        critical section
      s.V( );
        Remainder Section
    until false;
```

- Hand simulate execution of the above code

Resource Management in Semaphore

- Assume there are three identical printers
- A process that uses a printer executes the following code

```
var Semaphore s = new Semaphore(3);

Process:
    s.P( );
      find an available printer (you can definitely find one)
      use it
      return the printer
    s.V( );
```

Barrier Synchronization in Semaphore

```
var Semaphore s1 = new Seamphore(0);
    Semaphore s2 = new Seamphore(0);
```

```
T1: repeat
        Before the Barrier
      s2.V( );
      s1.P( );
        After the Barrier
    until false;
```

```
T2: repeat
        Before the Barrier
      s1.V( );
      s2.P( );
        After the Barrier
    until false;
```