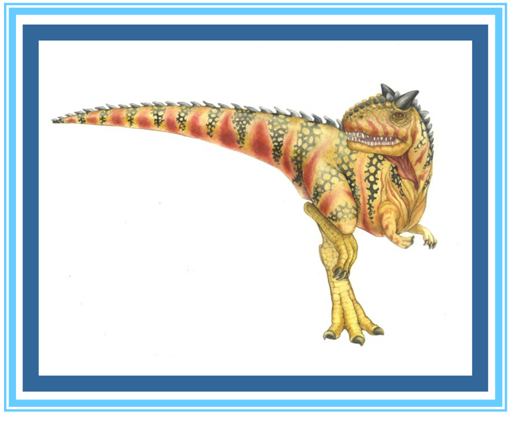
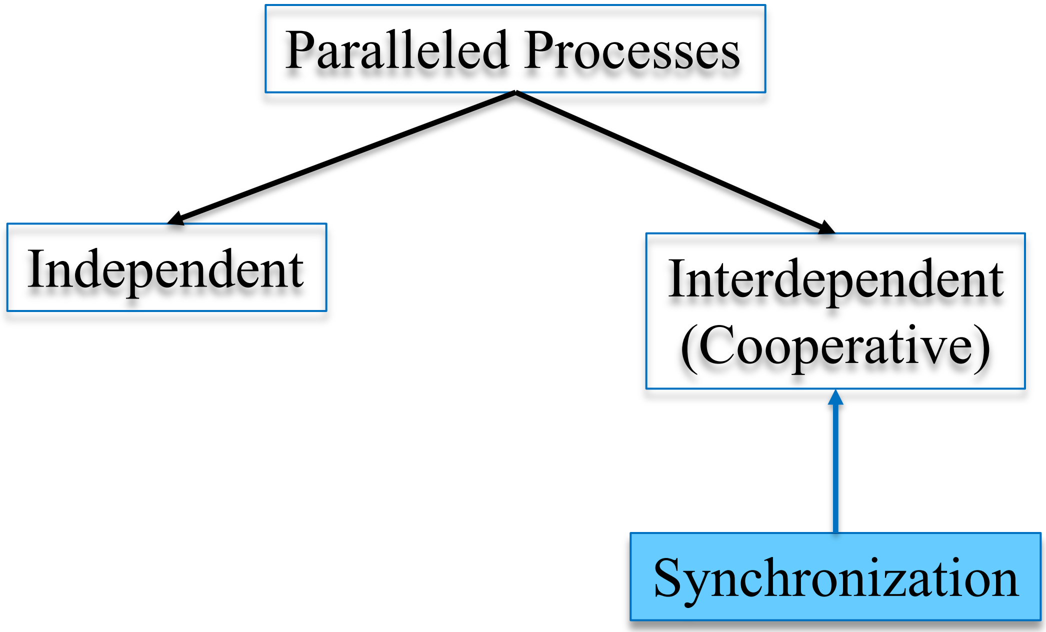
Chapter 6: Synchronization Tools



Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Background



Paralleled Processes
Independent
Interdependent
(Cooperative)
Synchronization

Producer

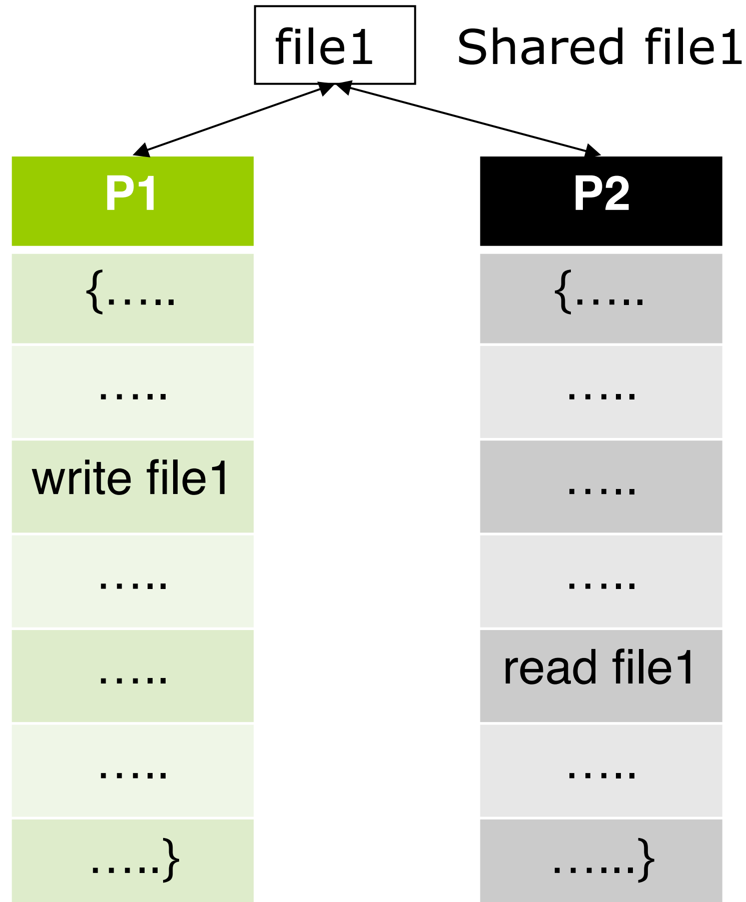
```
while (true) {
       /* produce an item in next produced */

       while (counter == BUFFER_SIZE)
              ; /* do nothing */
       buffer[in] = next_produced;
       in = (in + 1) % BUFFER_SIZE;
       counter++;
}
```

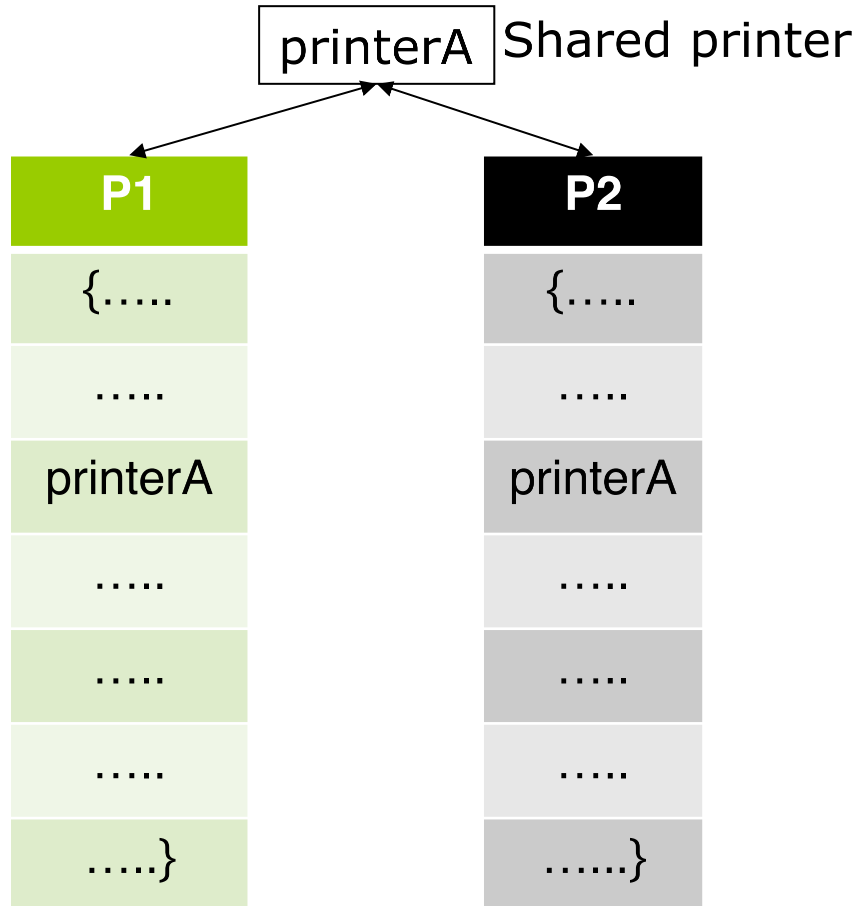
Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

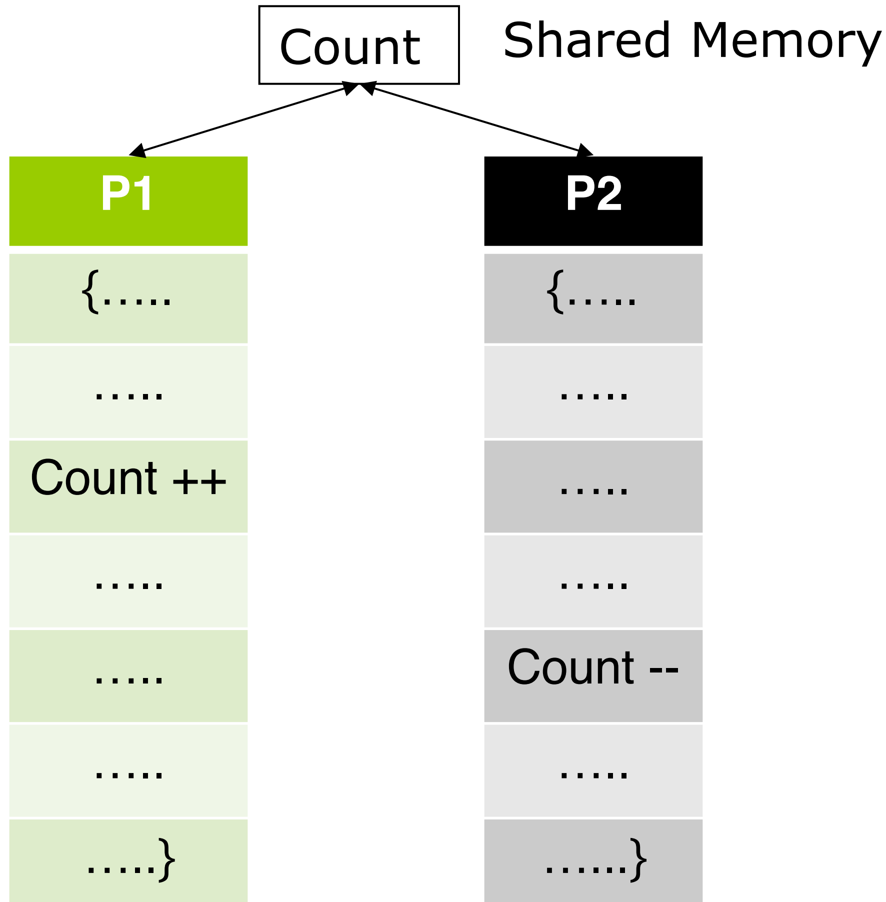
Introduction of Process Synchronization



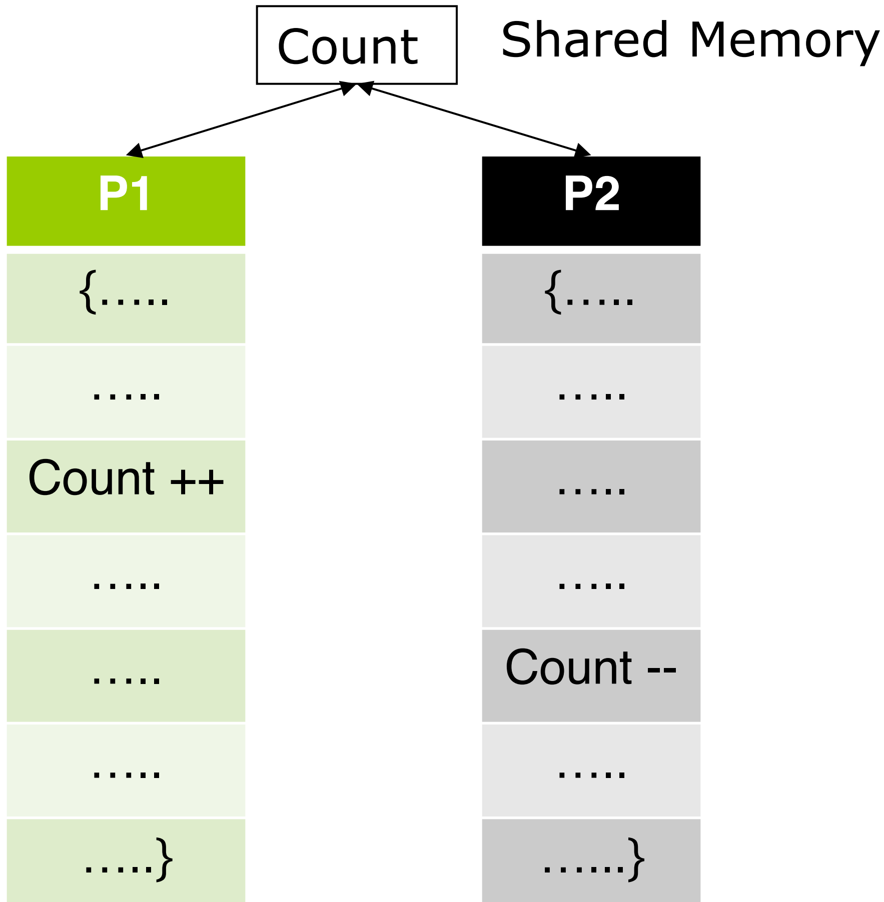
Introduction of Process Synchronization



Introduction of Process Synchronization



Race Condition



count =5
P1: count++ count = 6
P2: count -- count = 5

count =5
P2: count-- count = 4
P1: count++ count = 5

Race Condition

Count — Shared Memory

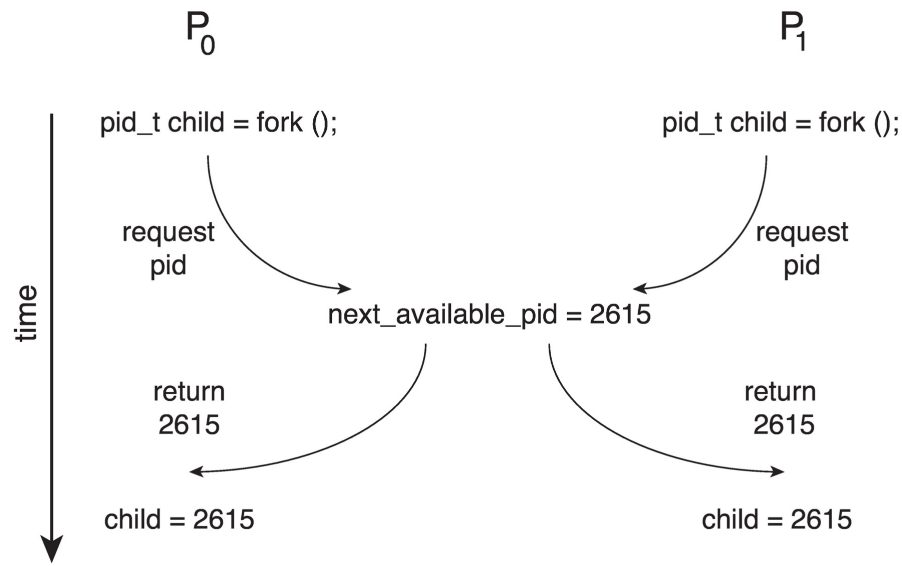
P1	P2
{…..	{…..
`register1=counter`	`register2 = counter`
`register1=register1+1`	`register2=register2-1`
`counter = register1`	`counter = register2`
…..}	…..}

Consider this execution interleaving with “counter = 5” initially:

S0: P1 execute `register1 = counter` {register1 = 5}
S1: P1 execute `register1 = register1 + 1` {register1 = 6}
S2: P2 execute `register2 = counter` {register2 = 5}
S3: P2 execute `register2 = register2 - 1` {register2 = 4}
S4: P1 execute `counter = register1` {counter = 6 }
S5: P2 execute `counter = register2` {counter = 4}

Race Condition

- Processes P_0 and P_1 are creating child processs using the **`fork()`** system call
- Race condition on kernel variable **`next_available_pid`** which represents the next available process identifier (pid)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!

Critical Section

P1	
{.....	
`register1=counter`	
`register1=register1+1`	CS
`counter = register1`	
.....	
.....	
.....}	

P2	
{.....	
`register2 = counter`	
`register2=register2-1`	CS
`counter = register2`	
......	
.....	
......}	

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {
        entry section           /* Vào critical section */

            critical section    /* Truy xuất dữ liệu chia sẻ */

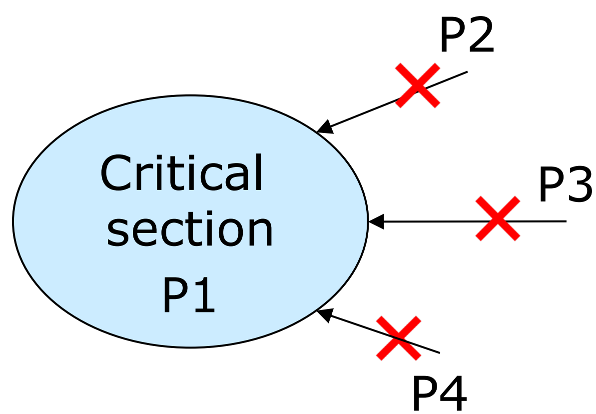
        exit section            /* Rời critical section */

            remainder section   /* Làm những việc còn lại */

} while (true);
```

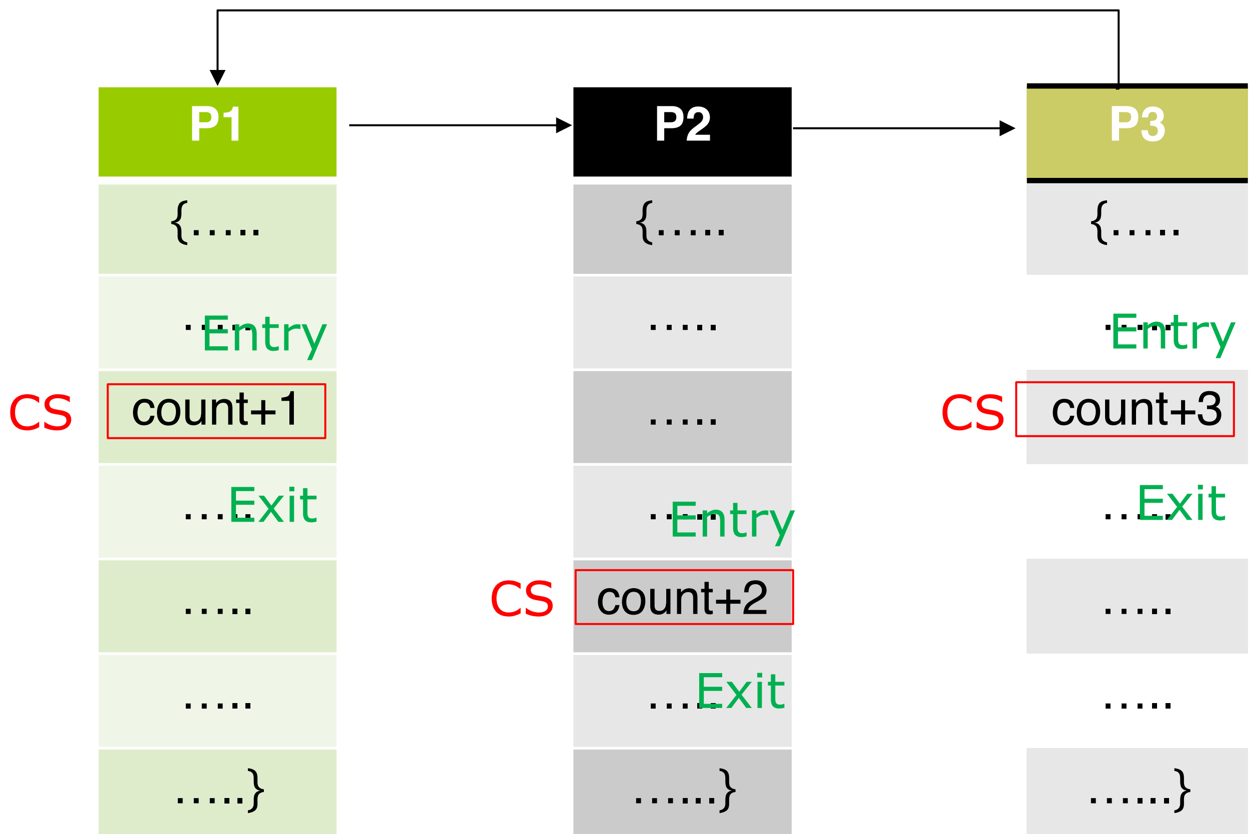
Solution to Critical-Section Problem

- **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



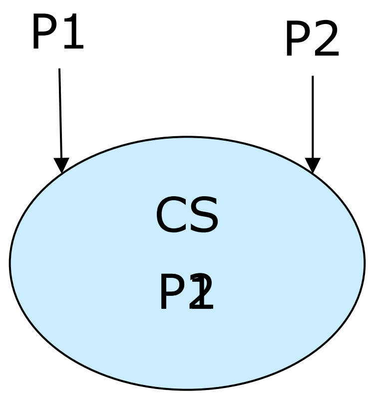
Solution to Critical-Section Problem

- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely



Solution to Critical-Section Problem

- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted



Process Solution-Algorithm1

❖ **Sử dụng việc kiểm tra luân phiên**

- Biến chia sẻ: int turn; /* khởi đầu turn = 0 */
- Nếu turn = i thì P_i được phép vào critical section, với i = 0 hay 1
- Process *Pi*

P_i
do{…..
while(turn!=i);
critical section
turn = j; /* j = 1-i*/
remainder section
}while(1)

Process Soluction-Algorithm1

turn

0

P_0
do{…..
entry: while(turn!=0);
critical section
exit: turn = 1;
remainder section
}while(1)

P_1
do{…..
while(turn!=1);
critical section
turn = 0;
remainder section
}while(1)

P0

CS
P0

P0 có RS nhỏ còn P1 có RS rất lớn ???

Process Solution-Algorithm2

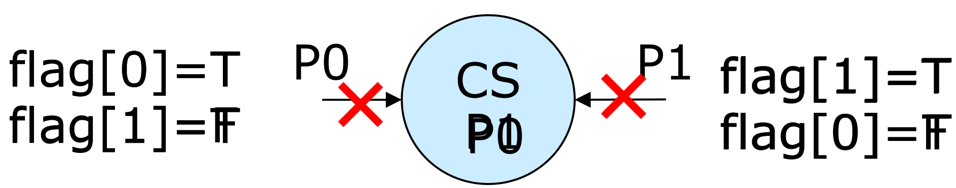
❖ **Sử dụng các biến cờ hiệu**

- Biến chia sẻ: boolean flag[i]; /* khởi đầu flag[0] = flag[1] = false */
- Nếu flag[i] = true thì P_i "sẵn sàng" vào critical section.
- Process P_i

P_i	
do{	
flag[i] = true	/* P_i "sẵn sàng" vào CS */
while(flag[j]);	
CS	
flag[i] = false;	
RS	
}while(1)	

Process Solution-Algorithm2

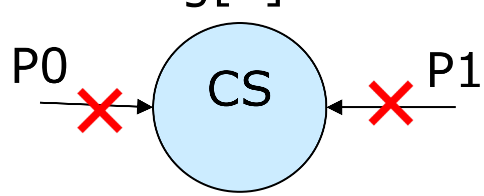
flag[0]	flag[1]
P $_0$	***P*** $_1$
{…..	{…..
flag[0]=T while(flag [1]);	flag[1]=T while(flag [0]);
CS	CS
flag[0] = F	flag[1] = F
RS	RS
…..}	…..}



Process Soluction-Algorithm2

flag[0]	flag[1]
P_0	P_1
{…..	{…..
flag[0]=T while(flag [1]);	flag[1]=T while(flag [0]);
CS	CS
…..}	…..}

flag[0]=T
flag[1]=T



Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process P_i

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     Pi enters CS only if:

     either flag[j] = false or turn = i
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Peterson's Soluction-Algorithm3

P0	P1
{…..	do{…..
flag[0] = T Turn = 1 while(flag[1])&&turn==1);	flag[1] = T Turn = 0 while(flag[1])&&turn==0);
critical section	Critical section
flag[0] = F	flag[1] = F
remainder section …..}	Remainder section …….}

flag[0]	flag[1]	turn]
F	F	
T	F	
F	T	
T	T	0
T	T	1

Peterson's Solution

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Memory barriers

2. Hardware instructions

3. Atomic variables

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - (Originally called **`P()`** and **`V()`**)
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
  typedef struct {
       int value;
       struct process *list;
  } semaphore;
  ```
  ```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal (mutex)`** …. **`wait (mutex)`**
  - **`wait (mutex)`** … **`wait (mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
- These – and others – are examples of what can occur when sempahores and other synchronization tools are used incorrectly.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

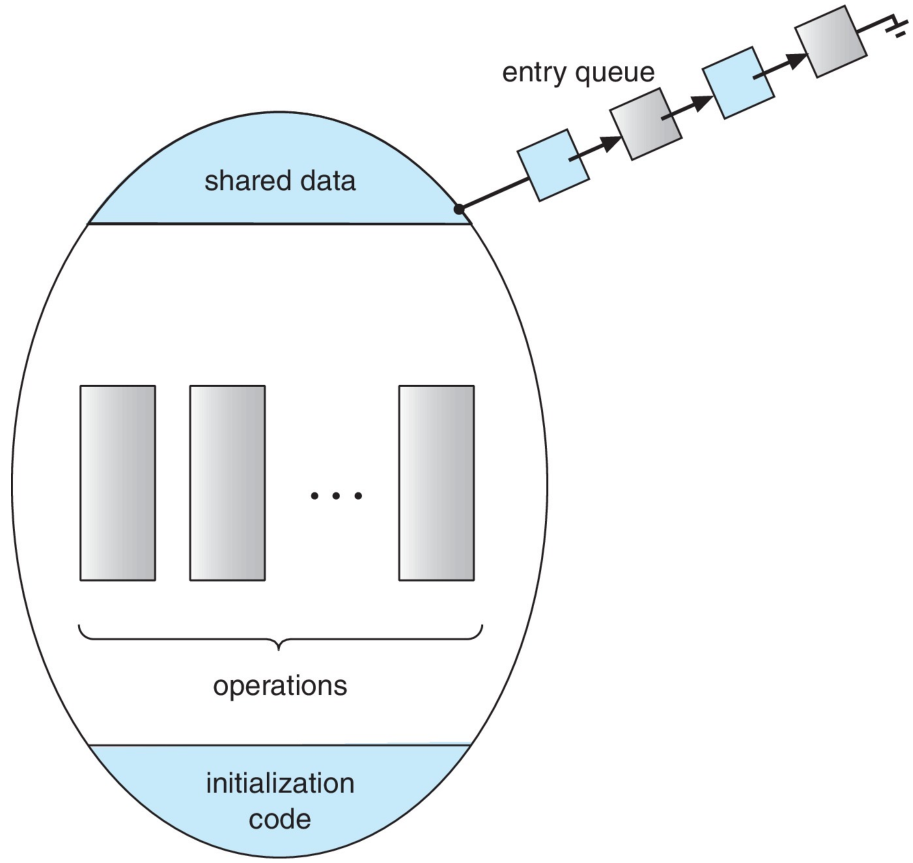
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```

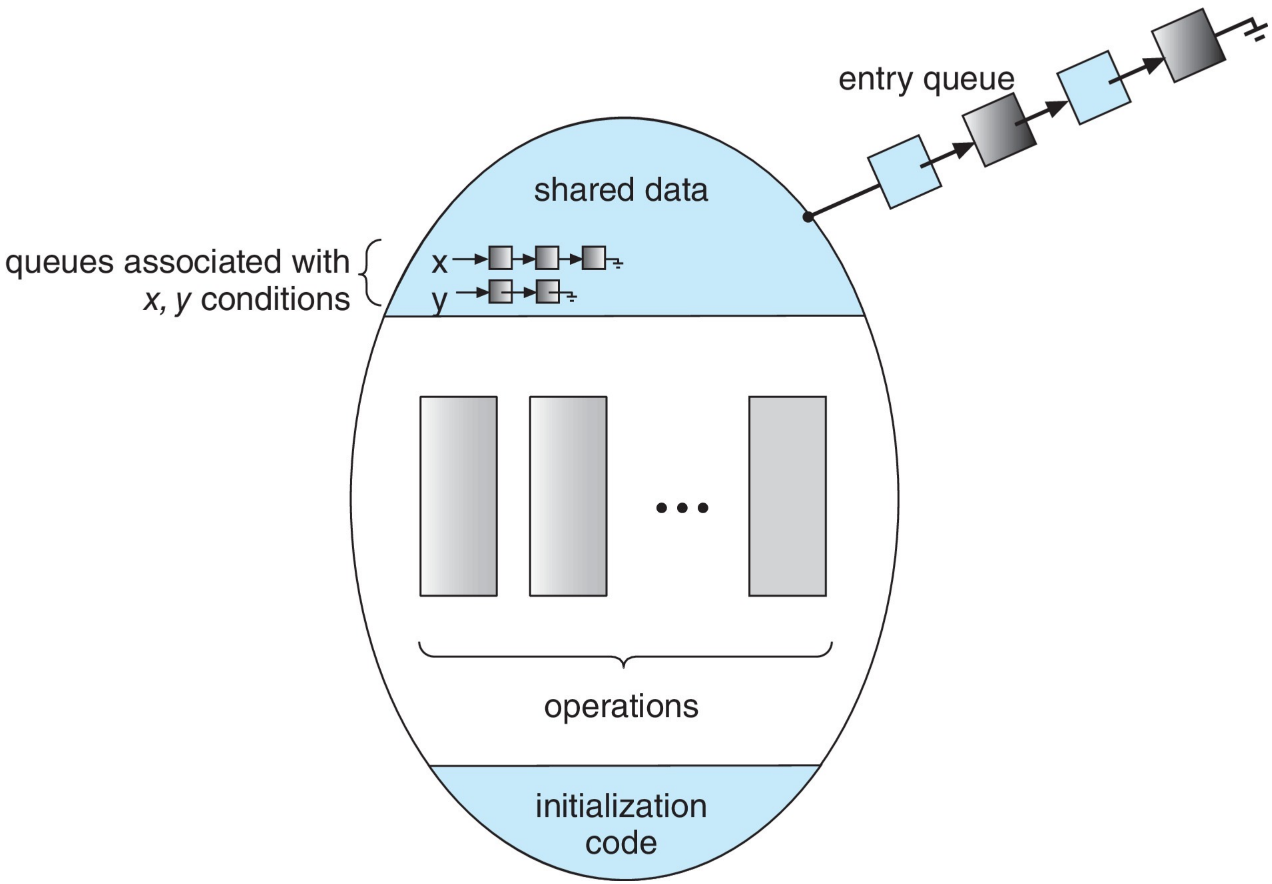
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each function ***F*** will be replaced by

```
wait(mutex);
     …
   body of F;
     …
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition variable **`x`**, and **`x.signal()`** is executed, which process should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form **`x.wait(c)`**
  - Where **`c`** is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
    ...
 access the resurce;
    ...

R.release;
```

- Where R is an instance of type **`ResourceAllocator`**

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = true;
   }
   void release() {
            busy = FALSE;
            x.signal();
   }
   initialization code() {
    busy = false;
   }
}
```

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

Liveness

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.

Liveness

- Other forms of deadlock:
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**

Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**. **P1** has the highest priority, **P2** the next highest, and **P3** the lowest. Assume a resouce **P3** is assigned a resource **R** that **P1** wants. Thus, **P1** must wait for **P3** to finish using the resource. However, **P2** becomes runnable and preempts **P3**. What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.

- To prevent this from occurring, a **priority inheritance protocol** is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.

End of Chapter 6