



OPERATING SYSTEMS

Process Synchronization



BACKGROUND

- Processes can execute concurrently or in parallel
- CPU scheduler switches rapidly between processes to provide concurrent execution
- A process may be interrupted at any point in its instruction stream
- Parallel execution, in which two instruction streams execute simultaneously on separate processing cores
- We will explain **how concurrent or parallel execution** can contribute to issues involving the **integrity of data shared** by several processes

PRODUCER-CONSUMER PROBLEM

- Modify the algorithm to remedy this deficiency - add an integer variable *counter*, initialized to 0
- *counter* is incremented every time we add a new item to the buffer
- decremented every time we remove one item from the buffer

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

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

DATA INTEGRITY PROBLEM

$register_1 = counter$
 $register_1 = register_1 + 1$
 $counter = register_1$

$register_2 = counter$
 $register_2 = register_2 - 1$
 $counter = register_2$

- “*counter++*” and “*counter--*” in machine language is like in the above.
- *register1* and *register2* is local CPU registers.
- Concurrent execution of “*counter++*” and “*counter--*” and allowing them to manipulate the counter variable create incorrect state.

T_0 :	<i>producer</i>	execute	$register_1 = counter$	$\{register_1 = 5\}$
T_1 :	<i>producer</i>	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2 :	<i>consumer</i>	execute	$register_2 = counter$	$\{register_2 = 5\}$
T_3 :	<i>consumer</i>	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4 :	<i>producer</i>	execute	$counter = register_1$	$\{counter = 6\}$
T_5 :	<i>consumer</i>	execute	$counter = register_2$	$\{counter = 4\}$

RACE CONDITION

- Several process **access and manipulate the same data** concurrently
- Outcome of the execution depends on the **particular order in which the access** takes place
- To guard against this condition –
 - Ensure that only one process at a time can manipulate the counter variable (shared data)
 - The processes should be **synchronized**



OPERATING SYSTEMS Critical Section



CRITICAL SECTION

- Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$.
- **Critical Section:** segment of code of each process, which may change common variables, update a table, write a file and so on.
- While one process execute its critical section, no other process can execute their own critical section.
- **Entry Section:** section of code implementing critical section execution request
- **Exit Section:** section of code exiting from critical section
- **Remainder section:** Remaining code of the program.

REQUIREMENTS OF SOLUTION TO CRITICAL SECTION PROBLEM

1. *Mutual exclusion:*

- ☐ If a process is executing its critical section, no other process can be executing in their critical sections.

2. *Progress:*

- ☐ No process is executing in its critical section
- ☐ Some process wish to enter their critical sections
- ☐ Only those, who are not executing in their remainder section can participate in deciding which will enter the CS.
- ☐ This selection cannot be postponed indefinitely.

3. *Bounded waiting:*

- ☐ Bound or Limit on number of times other process can enter their CS after a process has made request to enter its CS and the request is granted

CRITICAL SECTIONS IN OPERATING SYSTEMS

Two general approach to handle CS in Operating System –

1. *Preemptive kernel*: allows a process to be preempted while it is running in kernel mode
 2. *Non-preemptive kernel*: a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU
- Non-preemptive kernel is **free from race condition**
 - Preemptive kernel must be carefully designed to ensure that shared kernel data are free from race condition



OPERATING SYSTEMS

Peterson's solution for Critical Section Problem



SOFTWARE-BASED SOLUTION TO THE CRITICAL SECTION PROBLEM

- Known as “**Perterson’s Solution**”
- **Restricted to two processes** that alternate execution between their critical sections and remainder sections
- Peterson’s solution requires the two processes to **share two data** items:

int turn;

boolean flag[2];

- **turn**: indicates whose turn it is to enter its critical section
- **flag**: an array used to indicate if a process is ready to enter its critical section.

PETERSON'S SOLUTION

do {

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

critical section

```
flag[i] = false;
```

remainder section

} while (true);

The structure of process P_i in Peterson's solution.

- Process P_i first sets $flag[i]$ to be *true* and then sets *turn* to the value j , so that if the other process wants to enter its CS, it can do so.
- If both try to enter at the same time, *turn* will be both i and j at the same time, but, only one of these will last.
- The eventual value of *turn* determines which process will enter its critical section.

It may not work correctly on modern computer architecture as they perform basic machine-language instructions such as load and store.

Example

- Each Statement takes 2ms to execute, Process 1 gets executed first
- Context Switch will occur after 6ms
- Critical section contains 4 statements
- Remainder section contains 2 statements
- turn=0
- Flag[0] = FALSE, flag[1] = TRUE

do {

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

critical section


```
flag[i] = false;
```

remainder section

} while (true);


The structure of process P_i in Peterson's solution.

Process 0 (i = 0, j = 1)	Process 1 (i = 1, j = 0)
	flag[1] = TRUE turn = 0 While loop condition
flag[0] = TRUE turn = 1 Stuck in while loop	
	CS1 CS2 CS3
Stuck in while loop	
	CS4 flag[1] = FALSE RS1
While loop condition CS1 CS2	
	RS2
CS3 CS4 flag[0] = FALSE	
RS1 RS2	



OPERATING SYSTEMS

Hardware based solution for Critical Section Problem



HARDWARE-BASED SOLUTION TO THE CRITICAL SECTION PROBLEM

- More solutions to the critical-section problem using techniques ranging from hardware to software-based APIs
- These solutions are based on the premise of **locking** — protecting critical regions through the use of locks.
- In a single-processor environment CS problem can be solved by preventing interrupts from occurring while a shared variable is being modified.
- For multiprocessor environment, we need different measures.
- Modern computer systems allow to test and modify the content of a word or to swap the contents of two words atomically – which is uninterruptable unit. We can use *test_and_set()* and *compare_and_swap()* instructions.

TEST_AND_SET()

- Executed atomically
- Mutual exclusion can be implemented by initializing a Boolean variable lock to false

```
boolean test_and_set(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
  
    return rv;  
}
```

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
  
    /* remainder section */  
} while (true);
```


COMPARE_AND_SWAP()

- Mutual exclusion can be achieved by declaring a global variable *lock* and initializing it to *0*
- First process that invokes this instruction will set *lock* to *1* and no other process can execute CS until this process updates it to *0* after CS execution.

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
} while (true);
```

MUTEX LOCKS

- Operating-systems designers build software tools to solve CS problem.
- Simplest of these tools is “Mutex Lock” (Mutex = Mutual Exclusion)
- A process must acquire the lock before entering CS [*acquire()* function]
- A process must release the lock after exiting the CS [*release()* function]
- Mutex lock has a variable, *available* which indicates if the lock is available.

```
do {  
    acquire() {  
        while (!available)  
            ; /* busy wait */  
        available = false;;  
    }  
    release() {  
        available = true;  
    }  
} while (true);
```


acquire lock

critical section

release lock


remainder section

Figure 5.8 Solution to the critical-section problem using mutex locks.



OPERATING SYSTEMS

Semaphore



SEMAPHORE

- A semaphore S is an integer variable
- Is accessed only through two standard atomic operations: **wait()** and **signal()**.
- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption, i.e., **this operations are atomic**.

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

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

Types of Semaphores

- ❑ **Counting Semaphore:** The value can range over an unrestricted domain.
 - ❑ Used to control access to a given resource consisting of finite number of instances
 - ❑ Solves various synchronization problems.
- ❑ **Binary Semaphore:** The value can range only between 0 and 1. This behaves similar to Mutex Lock.

Counting Semaphore

- initialized to the number of resources available, $S = n$
- Each process that wishes to use a resource performs a wait() operation

$$S = S - 1$$

- When a process releases a resource, it performs a signal() operation

$$S = S + 1$$

- When S becomes 0 , all resources are being used
- processes that wish to use a resource will block until $S > 0$

Binary Semaphore - Synchronization

- P_1 has statement S_1
- P_2 has statement S_2
- We want to make sure that S_1 executes before S_2
- We can use a semaphore variable **sync** and initialize it to **0**

P1:

S_1 ;
signal(sync);

P2:

wait(sync);
 S_2 ;

Mutual Exclusion With Semaphores

- Binary Semaphores (mutex) can be used to solve CS problem.
- A semaphore variable (say mutex) can be shared by n processes and initialized to 1.
- Each process is structured as follows :

```
do{  
    wait (mutex);  
    //critical section  
    signal(mutex);  
    //remainder section  
}while (TRUE);
```


Deadlock & Starvation

- Two or more process can wait indefinitely for an event - **DEADLOCK !!!**
- It occurs because - two process depends on each other for causing an event in a specific manner

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

- **Starvation:** Processes wait indefinitely within the semaphore
- Occurs if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order



OPERATING SYSTEMS

Semaphore Implementation



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

- When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait
- Rather than this busy waiting, the process can block itself which places it into a waiting queue associated with the semaphore
- State of the process is switched to the waiting state and control is transferred to CPU scheduler which selects another process to execute.
- It will be restarted when some other process executes a `signal()` operation
- Restarted by a `wakeup()` operation that changes it from waiting state to ready state.

SEMAPHORE IMPLEMENTATION

Definition of a semaphore:

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

Definition of wait():

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

Definition of signal():

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