Operating Systems - Process Synchronization

Mridul Sankar Barik

Jadavpur University

2024

Slide Credits

- Most of the slides are adapted from the companion lecture slides of the text book by Avi Silberschatz, Peter Baer Galvin, Greg Gagne

Producer Consumer Problem I

- This is one example of cooperating processes
- A producer process produces information which is consumed by a consumer process
  - A print process produces characters which is consumed by the printer driver
  - A compiler produces assembly code which is consumed by an assembler
- To run concurrently there must be a buffer
- The producer and consumer processes must be synchronized so that
  - Consumer does not try to consume an item that has not yet been produced
- Two variations
  - Unbounded Buffer
    - Consumer may have to wait for new items
    - Producer can always produce new items
  - Bounded Buffer
    - Consumer must wait if the buffer is empty
    - Producer must wait if the buffer is full

Producer Consumer Problem II

Data Structures

```
/*The shared buffer is implemented as a circular array with two logical pointer in and out*/

#define BUFFER_SIZE 10
typedef struct{
...
}item;

item buffer[BUFFER_SIZE];

int in=0;          /*in points to the next free position in the buffer*/
int out=0;         /*out points to the first full position in the buffer*/

int counter=0;     /*incremented every time an item is produced and decremented every time
                   an item is consumed*/
```

Producer

```
/*The producer produces a new item in the local variable
nextProduced*/

while(true){
    while(counter == BUFFER_SIZE);
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
    counter++;
}
```

Consumer

```
/*The consumer has a local variable nextConsumed in
which the item to be consumed is stored*/

while(true){
    while(counter == 0);
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    counter--;
}
```

Producer Consumer Problem III

- The producer and consumer routines may not function correctly when executed concurrently
- Example
  - Current value of counter is 5
  - Producer and Consumer process execute the statements `counter++` and `counter--` concurrently
  - After execution the value of variable counter may be 4, 5 or 6
  - `counter++` may be implemented as
    1. `register1=counter`
    2. `register1=register1+1`
    3. `counter=register1`
  - `counter--` may be implemented as
    1. `register2=counter`
    2. `register2=register2-1`
    3. `counter=register2`

Producer Consumer Problem IV

- Concurrent execution of `counter++` and `counter--` is equivalent to sequential execution of lower level statements in some arbitrary manner
  - T1 producer execute `register1=counter` {`register1=5`}
  - T2 producer execute `register1=register1+1` {`register1=6`}
  - T3 consumer execute `register2=counter` {`register2=5`}
  - T4 consumer execute `register2=register2-1` {`register2=4`}
  - T5 producer execute `counter=register1` {`counter=6`}
  - T6 consumer execute `counter=register2` {`counter=4`}
- Race Condition
  - When several processes access and manipulate the same data concurrently and the outcome of the execution depends on particular order in which the access takes place
  - To guard against race condition, we need to ensure that only one process at a time can be manipulating shared data

Critical Section Problem I

- Consider a system of n processes $P_0, P_1, P_2, \ldots, P_{n-1}$
- Each process has a segment of code called critical section where it accesses shared data (changing common variables, updating table, writing file, etc)
- The execution of critical sections by the processes is mutually exclusive in time
- In entry section each process requests permission to enter its critical section

General Structure of Process P_i

```
do {
    < entry section >
    < critical section >
    < exit section >
    < remainder section >
} while(true);
```

Critical Section Problem II

- A solution to the critical section problem must satisfy the requirements
  - Mutual Exclusion
    - If a process is executing in its critical section then no other process can execute it its critical section
  - Progress
    - If no process is executing in its critical section and some processes wish to enter its critical sections, then those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely
  - Bounded Waiting
    - There exists a bound 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

Two Process Solutions I

- Algorithms applicable to two processes P_0 and P_1

Algorithm 1

- Two processes share a common integer variable `turn` initialized to 0 (or 1)

```
do{
     while(turn != i);
     < critical section >
     turn = j;
     < remainder section >
}while(true);
```

- Ensures mutual exclusion
- Does not satisfy progress requirement since it requires strict alternation of processes in the execution of critical section

Two Process Solutions II

Algorithm 2

- All `flag[i]`'s are initialized to `false`
- `flag[i]=true` indicates that process P_i is ready to enter its critical section

```
do{
     flag[i] = true;
     while (flag[j]);
     < critical section >
     flag[i] = false;
     < remainder section >
}while(true);
```

- Ensures mutual exclusion
- Does not satisfy progress requirement
- Following sequence of execution leaves the two processes in deadlock
  - T0: P_0 sets `flag[0] = true`
  - T1: P_1 sets `flag[1] = true`

Two Process Solutions III

Algorithm 3

- Initially `flag[0]=flag[1]=false`, and `turn` is either 0 or 1

```
do{
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j);
     < critical section >
     flag[i] = false;
     < remainder section >
}while(true);
```

- This algorithm satisfies all three requirements of critical section problem
- Also known as Peterson's solution

Multiple Process Solution

Bakery Algorithm

- The common data structures are:

```
boolean choosing[n];
int number[n];
```

- Initially these data structures are initialized to `false` and 0, respectively
- For convenience, we define
  - `(a, b)<(c, d)` if `a<c` or if `a==c` and `b<d`
  - `max(`a_0`, `a_1`, `a_2`, ..., `a_{n-1}`)` is a number `k`, such that `k`$\geq a_i$, for $i = 0, 1, 2, 3, \ldots, n-1$

Structure of process P_i

```
do{
    choosing[i]=true;
    number[i]=max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i]=false;
    for(j=0; j<n; j++){
        while(choosing[j]);
        while((number[j]!=0)&&((number[j], j)<(number[i], i)));
    }
    < critical section >
    number[i]=0;
    < remainder section >
}while(true);
```

Synchronization Hardware I

- Critical section problem can be solved in a uniprocessor system by disabling interrupts to occur while shared variable is being updated
- In multiprocessor system disabling interrupts can be time consuming
- Many processors provide instructions to atomically
  - Test and modify the content of a word
  - Swap the contents two words
- Definition of `TestAndSet` instruction

```
boolean TestAndSet(boolean *target){
    boolean rv = *target;
    *target = True;
    return rv;
}
```

Synchronization Hardware II

- If a machine supports `TestAndSet` instruction, then mutual exclusion can be implemented by having a shared variable `lock` set to `false`

```
do{
     while(TestAndSet(lock));
     < critical section >
     lock=false;
     < remainder section >
}while(1);
```

Synchronization Hardware III

- Definition of swap instruction

```
boolean swap(boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

- If a machine supports swap instruction then mutual exclusion can be implemented by having
  - A global variable lock initialized to false
  - A local variable key maintained by each process

```
do{
    key = true;
    while(key == true)
    swap(lock, key);
    < critical section >
    lock=false;
    < remainder section >
}while(1);
```

- This algorithm does not support bounded-waiting requirement

Synchronization Hardware IV

- Algorithm that satisfies all critical section requirements
  - The common data structures are

```
boolean waiting[n];
boolean lock;
```

```
do{
    waiting[i]=true;
    key=true;
    while(waiting[i] && key)
        key=TestAndSet(lock);
    waiting[i] = false;
    < critical section >
    j=(i+1)%n;
    while((j!=i) && !waiting[j])
        j=(j+1)%n;
    if(j==i)
        lock=false;
    else
        waiting[j]=false;
    < remainder section >
}while(1);
```

Semaphores I

- OS kernel provides synchronization tool semaphore to solve critical section problem
- A semaphore is an integer variable that, apart from initialization can be accessed only through two standard atomic operations `wait()` and `signal()`
  - Originally called `P()` and `V()` operations

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

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

- `wait` and `signal` operations are atomic

Semaphores II

- Semaphores can be used to solve n process critical section problem
- The *n* processes share a semaphore mutex

```
do{
    wait(mutex);
    < critical section >
    signal(mutex);
    < remainder section >
}while(1);
```

- **Counting semaphore**: integer value can range over an unrestricted domain
- **Binary semaphore**: integer value can range only between 0 and 1
- Semaphores can also be used to solve various synchronization problems

Semaphore Implementation I

- Problem with solution of critical section problem using semaphores
  - Busy Waiting
    - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry section
    - Wastes CPU cycles those can otherwise be used by other processes productively
    - This type of semaphores are called spinlock (process spins while waiting for the lock)
    - Spinlock is advantageous in multiprocessor systems as no context switch is required while waiting for a lock
  - Solution
    - When a process executes wait and finds semaphore value not positive, it blocks itself
    - A blocking process is placed into a waiting queue associated with the semaphore
    - When a process executes signal, it wakes up a process from the waiting queue associated with that semaphore

Semaphore Implementation II

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

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

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

Deadlock and Starvation

- **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

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

P_1

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

- **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

Classical Problems of Synchronization I

- The **Bounded Buffer Producer Consumer Problem**
- Assumptions
  - The shared pool consists of n buffers
  - `mutex` semaphore provides exclusive access to the buffer and is initialized to 1
  - The `empty` (initialized to n) and `full` (initialized to 0) semaphore count the number of empty and full buffers

Producer

```
do{
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
}while(1);
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    consume the item in nextc
}while(1);
```

Classical Problems of Synchronization II

- The **Readers-Writers Problem**
  - Two readers can access the shared object simultaneously
  - One writer and some other process cannot access the shared object simultaneously
  - Shared data structures
    `semaphore mutex, wrt;`
    `int readcount;`
  - `mutex` and `wrt` are initialized to 1; `readcount` is initialized to 0

Reader

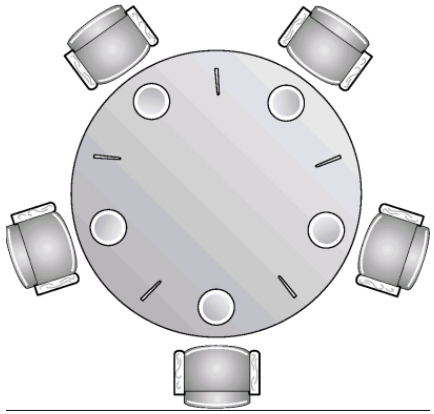
```
do{
    wait(mutex);
    readcount++;
    if(readcount==1)
        wait(wrt);
    signal(mutex);
    ...
    reading is performed
    ...
    wait(mutex);
    readcount--;
    if(readcount==0)
        signal(wrt);
    signal(mutex);
}while(1);
```

Writer

```
do{
    ...
    wait(wrt);
    ...
    writing is performed
    ...
    signal(wrt);
    ...
}while(1);
```

Classical Problems of Synchronization III

- The **Dining Philosophers Problem**
- Consider five philosophers who spend their lives thinking and eating
- Philosophers share a common circular table surrounded by five chairs
- The table is laid with five single chopsticks and at the center of the table is a bowel of rice
- When a philosopher thinks she doesn't interact with others
- When a philosopher is hungry, picks up two chopsticks closest to her
- A philosopher can pick up only one chopstick at a time
- When she finishes eating, she puts down both of her chopsticks and starts thinking again



Structre of Philsopher *i*

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5];
    ...
    <eat>
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5];
    ...
    <think>
    ...
}while(1);
```

Classical Problems of Synchronization IV

- This solution of the Dining Philosophers Problem is not deadlock free
- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution - an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.