CS415 Module 4 Part A - Concurrency and Synchronization

Athens State University

Outline

Contents

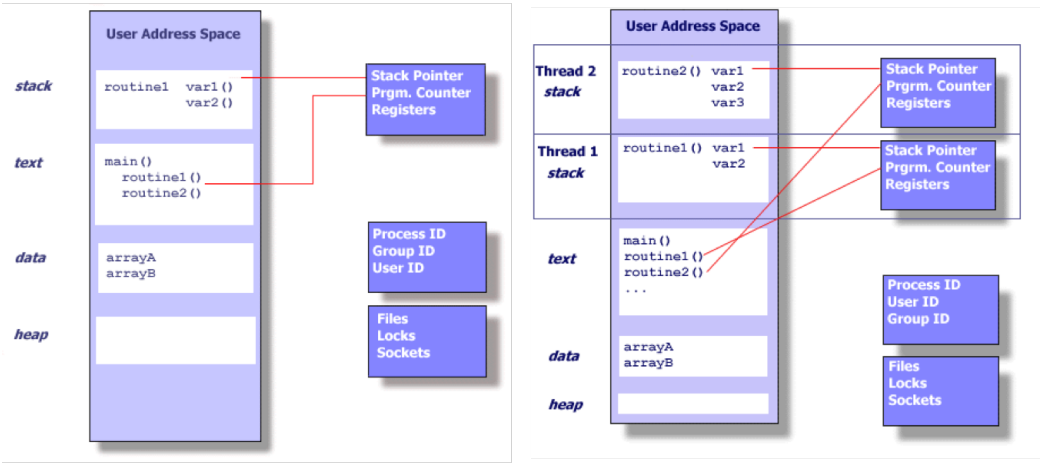
1 The Need for Concurrency

Multiple Processes

Operating systems have to be concerned with the following when managing processes and threads:

- Multiprogramming
  - Managing multiple processes on a uniprocessor
- Multiprocessing
  - Managing multiple processes on a multiprocessor
- Distributed processing
  - Managing multiple processes on multiple distributed processors (either uni-or multiprocessor)

Process vs. thread



Concurrency

- Multiple Applications
  - Invented to allow processing time to be shared among active applications
- Structured Applications
  - Extension of modular design and structured programming
- Operating System Structure
  - OS are implemented as a set of processes and threads

Key Terms

Atomic operation A function or action implemented as a sequence of one or more instructions that appears to be indivisible

Critical section A section of code within a process or thread that requires access to shared resources and must not be executed while other process is executing in the same code

Mutual exclusions The requirement that when one process thread is in a critical section, then no other process or thread can be in a critical section that accesses the same resources

Atomic operations are guaranteed to execute as a group, or not execute at all. Atomicity guarantees isolation from concurrent processes or threads.

Key Terms

Deadlock A situation in which two or more processes or threads are unable to proceed because each is waiting for one of the others to do something

Livelock A situation in which two or more processes continuously change their state in response to chages in the other processes without doing any useful work

Race Condition A situation in which multiple processes or threads read and write a shared data item and final result depends upon the relative timing of their execution

Starvation A situation in which a runnable process is overlooked indefinitely by the scheduler

Difficulties of Concurrency

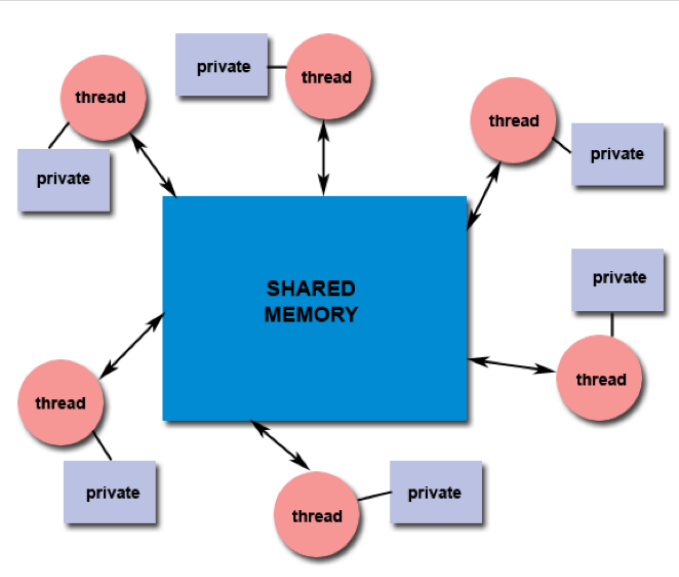
- Sharing of global resources
- Difficult for the OS to manage allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

2 Processes, Threads, and Concurrency

Threads Share Resources

- Changes made by one thread to shared system resources will be seen by all threads
  - Consider what happens when you close a file
- Two pointers have the same value point to the same data
- Reading and writing to the same memory location is possibile
  - Requires explicit synchronization by the developer

Shared Memory Model



- All threads have access to the same global memory
- Threads also have their own private data
- Programmers have to make certain threads don't stomp on each other

How Processes and Threads Interacte

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Resource Competition

Concurrent processes come into conflict when they compete for the same resource

Fixing competing processes requires solving three problems:

- Address the need for mutual exclusion
- Avoid deadlock
- Prevent starvation

Concurrent processes come into conflict with each other when they are competing for the use of the same resource. In its pure form, we can describe the situation as follows. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of other processes, and each is to be unaffected by the execution of the other processes. It follows from this that each process should leave the state of any resource that it uses unaffected. Examples of resources include I/O devices, memory, processor time, and the clock.

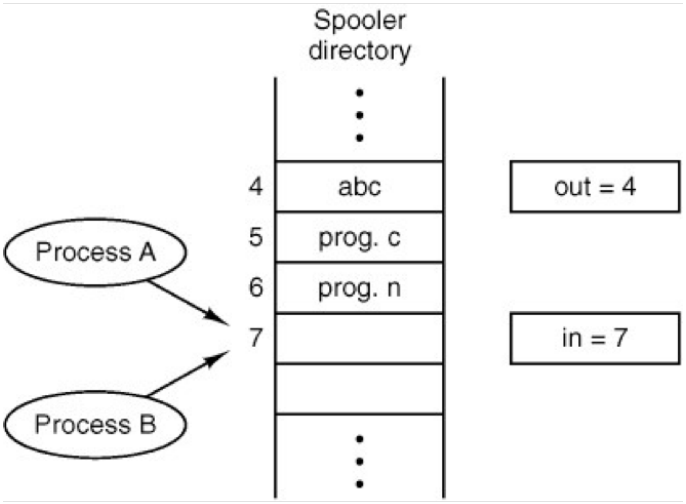
There is no exchange of information between the competing processes. However, the execution of one process may affect the behavior of competing processes. In particular, if two processes both wish access to a single resource, then one process will be allocated that resource by the OS, and the other will have to wait. Therefore, the process that is denied access will be slowed down. In an extreme case, the blocked process may never get access to the resource and hence will never terminate successfully.

In the case of competing processes three control problems must be faced. First is the need for mutual exclusion . Suppose two or more processes require access to a single non-sharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a critical resource , and the portion of the program that uses it as a critical section of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the OS to understand and enforce this restriction because the detailed requirements may not be obvious. In the case of the printer, for example, we want any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

The enforcement of mutual exclusion creates two additional control problems. One is that of deadlock . For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the OS assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.

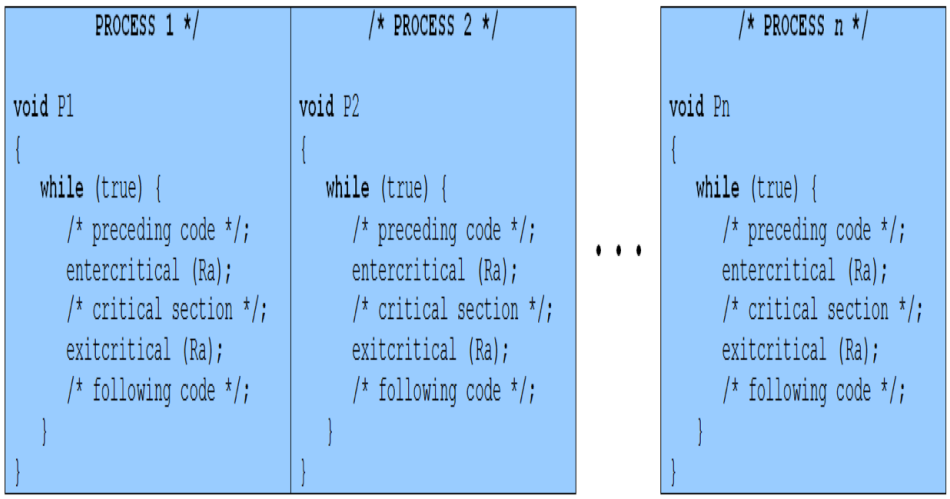
A final control problem is starvation . Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

Race Condition



- Occurs when multiple threads read and write to the same data items
- The final order depends on the order of execution
- The "loser" of the race is the process that updates last and so determines the final value of a variable

Mutual Exclusions



Control of competition inevitably involves the OS because it is the OS that allocates resources. In addition, the processes themselves will need to be able to express the requirement for mutual exclusion in some fashion, such as locking a resource prior to its use. Any solution will involve some support from the OS, such as the provision of the locking facility.

The figure in the slide illustrates the mutual exclusion mechanism in abstract terms using pseudo-code. There are n processes to be executed concurrently. Each process includes (1) a critical section that operates on some resource R_a, and (2) additional code preceding and following the critical section that does not involve access to R_a. Because all processes access the same resource R_a, it is desired that only one process at a time be in its critical section.

To enforce mutual exclusion, two functions are provided: `entercritical` and `exitcritical` . Each function takes as an argument the name of the resource that is the subject of competition. Any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait.

It remains to examine specific mechanisms for providing the functions `entercritical` and `exitcritical` . For the moment, we defer this issue while we consider the other cases of process interaction.

Requirements for Mutual Exclusion

- Must be enforced
- A process that halts must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes A process remains inside its critical section for a finite time only

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.

3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.

4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

5. No assumptions are made about relative process speeds or number of processors. 6. A process remains inside its critical section for a finite time only.

There are a number of ways in which the requirements for mutual exclusion can be satisfied. One approach is to leave the responsibility with the processes that wish to execute concurrently. Processes, whether they are system programs or application programs, would be required to coordinate with one another to enforce mutual exclusion, with no support from the programming language or the OS. We can refer to these as software approaches. Although this approach is prone to high processing overhead and bugs, it is nevertheless useful to examine such approaches to gain a better understanding of the complexity of concurrent processing. A second approach involves the use of special purpose machine instructions. These have the advantage of reducing overhead but nevertheless will be shown to be unattractive as a general-purpose solution. A third approach is to provide some level of support within the OS or a programming language.

Hardware Support for Mutual Exclusion

Interrupt Disabling:

- Must be a uniprocessor system
- Disabling interrupts guarantees mutual exclusion

Disadvantages:

- Noticeably degrades system performance
- Does not work in a multiprocessor/multicore environment

Hardware Support for Mutual Exclusion

- Requires special machine instructions designed to guarantee atomicity
- The Compare & Swap Instruction
  - Also known as a "compare and exchange instruction"
  - A *compare* is made between a memory value and a test value
  - If the values are the ***same***, then *swap* the values
  - Hardware guaranteed atomic instruction

The Compare & Swap the instruction checks a memory location (*word) against a test value (testval). If the memory location's current value is testval, it is replaced with newval ; otherwise it is left unchanged. The old memory value is always returned; thus, the memory location has been updated if the returned value is the same as the test value. This atomic instruction therefore has two parts: A compare is made between a memory value and a test value; if the values are the same, a swap occurs. The entire compare&swap function is carried out atomically—that is, it is not subject to interruption.

Another version of this instruction returns a Boolean value: true if the swap occurred; false otherwise. Some version of this instruction is available on nearly all processor families (x86, x86_64, arm, sparc, /390, etc.), and most operating systems use this instruction for support of concurrency.

Mutual Exclusion Algorithm Using Compare and Swap

Algorithm 1: Mutual Exclusion Using Compare and Swape

```
   Input: A set of n processes
1  Procedure i: int
2      while true do
3          while (compAndSwap(bolt,0,1) = 1) do
4          bolt ← 1;
5          Do the rest of your code;
6  Procedure
7      bolt ← 0;
8      parbegin(P(1), P(2), ..., P(n));
```

The algorithm implements a mutual exclusion protocol based on the use of the compare & swap instruction. A shared variable `bolt` is initialized to 0. The only process that may enter its critical section is one that finds bolt equal to 0. All other processes attempting to enter their critical section go into a busy waiting mode. The term busy waiting , or spin waiting , refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance. When a process leaves its critical section, it resets `bolt` to 0; at this point one and only one of the waiting processes is granted access to its critical section. The choice of process depends on which process happens to execute the compare & swap instruction next.

Special Machine Instruction: Advantages

- Applicable to any number of processes either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Special Machine Instruction: Disadvantages

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock is possible

3 Operating System and Programming Language Concurrency Constructs

`Pthreads` Mutexes

```c
#include <pthread.h>
#include <stdio.h>
#include <math.h>

#define ITERATIONS 10000

// A shared mutex
pthread_mutex_t mutex;
```

```
double target;

void* opponent(void *arg)
{
    for(int i = 0; i < ITERATIONS; ++i)
    {
        // Lock the mutex
        pthread_mutex_lock(&mutex);
        target -= target * 2 + tan(target);
        // Unlock the mutex
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}
```

Pthreads Mutexes

```
int main(int argc, char **argv)
{
    pthread_t other;

    target = 5.0;

    // Initialize the mutex
    if(pthread_mutex_init(&mutex, NULL))
    {
        printf("Unable to initialize a mutex\n");
        return -1;
    }

    if(pthread_create(&other, NULL, &opponent, NULL))
    {
        printf("Unable to spawn thread\n");
        return -1;
    }
    for(int i = 0; i < ITERATIONS; ++i) {
        pthread_mutex_lock(&mutex);
        target += target * 2 + tan(target);
        pthread_mutex_unlock(&mutex);
    }
    if(pthread_join(other, NULL)) {
        printf("Could not join thread\n");
        return -1;
    }
    // Clean up the mutex
    pthread_mutex_destroy(&mutex);
    printf("Result: %f\n", target);
    return 0;
}
```

Semaphore

- **Semaphore**: An integer-valued data structure that can only be modified by one of three operations:
    1. May be initialized to a nonnegative integer value
    2. The `semWait()` operation decrements the value

3. The `semSignal()` operation increments the value

- **NOTE**: There is no way to inspect or manipulate semaphores other than these operations!

The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals. For signaling, special variables called semaphores are used. To transmit a signal via semaphore s , a process executes the primitive semSignal(s) . To receive a signal via semaphore s , a process executes the primitive semWait(s) ; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:

1. A semaphore may be initialized to a nonnegative integer value.
2. The semWait operation decrements the semaphore value.
   - If the value becomes negative, then the process executing the semWait is blocked.
   - Otherwise, the process continues execution.
3. The semSignal operation increments the semaphore value.
   - If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Other than these three operations, there is no way to inspect or manipulate semaphores.

We explain these operations as follows. To begin, the semaphore has a zero or positive value. If the value is positive, that value equals the number of processes that can issue a wait and immediately continue to execute. If the value is zero, either by initialization or because a number of processes equal to the initial semaphore value have issued a wait, the next process to issue a wait is blocked, and the semaphore value goes negative. Each subsequent wait drives the semaphore value further into minus territory. The negative value equals the number of processes waiting to be unblocked. Each signal unblocks one of the waiting processes when the semaphore value is negative.

Implications of The Definition of Semaphores

This is no way to know before a process decrements a semaphore whether or not it will block.

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

Binary Semaphores

A binary semaphore is defined by the following three operations:

1. A binary semaphore may be initialized to 0 or 1.
2. The `semWaitB()` operation checks the semaphore value.
   - If the value is zero, then the process executing the `semWaitB()` is blocked.
   - If the value is one, then the value is changed to zero and the process continues execution.

3. The `semSignalB()` operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0).
    - If so, then a process blocked by a `semWaitB()` operation is unblocked.
    - If no processes are blocked, then the value of the semaphore is set to one.

In principle, it should be easier to implement the binary semaphore, and it can be shown that it has the same expressive power as the general semaphore. To contrast the two types of semaphores, the nonbinary semaphore is often referred to as either a counting semaphore or a general semaphore.

3.1 POSIX Semaphores

POSIX Semaphores

[fragile]

```
sem_t sem;

int sem_init(sem_t sem, int pshared, unsigned int value);

int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);

int sem_destroy(sem_t *sem);
```

Using Semaphores with `Pthreads`

```
#include <semaphore.h>
#include <pthread.h>
#define THREADS 20
sem_t OKToBuyMilk;
int milkAvailable;

void *buyer(void *arg) {
  sem_wait(&OKToBuyMilk);
  if (!milkAvailable) {
     // Buy some milk
     ++milkAvailable;
  }
  sem_post(&OKToBuyMilk);
  return NULL:
}
```

Using Semaphores with `Pthreads`

```
int main(int argc, char *argv[]) {
  pthread_t threads[THEADS];
  milk_available = 0;
  // Init semaphore to 1, second argument set to 0
  // indicates sharing among threads rather processes
  if (sem_init(&OKToBuyMilke, 0, 1)) {
    printf("Could not init semaphore\n");
    return -1;
  }
  for (int i = 0; i < THREADS; ++i) {
      if (pthread_create(&threads[i], NULL, &buyer, NULL)) {
```

```
            printf("Create thread %d failed\n",i);
            return -1;
        }
    }
    for (int i = 0; i < THREADS; ++i) {
        if (pthread_join(threads[i], NULL)) {
            printf("Could not join thread %d\n", i)
            return -1;
        }
    }
    sem_destroy(&OKToBuyMilk);
    printf("Total milk: %d\n", milkAvailable);
    return 0;
}
```

Barriers

- A barrier for a group of threads is a synchronization primitive that forces threads to wait until all threads in the group reach the barrier
- The basic form of a barrier uses two variables to keep track of its state
  - A lock variable saying whether the barrier is open or closed
  - An integer keeping track of the total number of threads at the barrier

`Pthreads` Barriers

```
#define _XOPEN_SOURCE 600
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define ROWS 10000
#define COLS 10000
#define THREADS 10

double initial_matrix[ROWS][COLS];
double final_matrix[ROWS][COLS];
// Barrier variable
pthread_barrier_t barr;
extern void DotProduct(int row, int col,
                       double source[ROWS][COLS],
                       double destination[ROWS][COLS]);
extern double determinant(double matrix[ROWS][COLS]);
```

`Pthreads` Barriers

```
void * entry_point(void *arg) {
    int rank = (int)arg;
    for(int row = rank * ROWS / THREADS; row < (rank + 1) * THREADS; ++row)
        for(int col = 0; col < COLS; ++col) {
            DotProduct(row, col, initial_matrix, final_matrix);
        }
    // Synchronization point
    int rc = pthread_barrier_wait(&barr);
    if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD)
    {
```

```
        printf("Could not wait on barrier\n");
        exit(-1);
    }
    for(int row = rank * ROWS / THREADS; row < (rank + 1) * THREADS; ++row)
        for(int col = 0; col < COLS; ++col)
            DotProduct(row, col, final_matrix, initial_matrix);
}
```

`Pthreads` Barriers

```
int main(int argc, char *argv[]) {
    pthread_t thr[THREADS];
    // Barrier initialization
    if(pthread_barrier_init(&barr, NULL, THREADS)) {
        printf("Could not create a barrier\n");
        return -1;
    }
    for(int i = 0; i < THREADS; ++i) {
        if(pthread_create(&thr[i], NULL, &entry_point, (void*)i)) {
            printf("Could not create thread %d\n", i);
            return -1;
        }
    }
    for(int i = 0; i < THREADS; ++i) {
        if(pthread_join(thr[i], NULL)) {
            printf("Could not join thread %d\n", i);
            return -1;
        }
    }
    double det = Determinant(initial_matrix);
    printf("The determinant of M^4 = %f\n", det);
    return 0;
}
```

4 Key Points

Key Points

- What is concurrency?
- What is mutual exclusion?
  - Hardware support for mutual exclusion
- OS and library support for synchronization primitives