Basic Set of Material to Cover in Lecture on Chapter Five of 9th Edition of Silberschatz. Process Synchronization

PROCESS SYNCHRONIZATION

OBJECTIVES

* introduce the critical section problem, whose solutions can be used to ensure the consistency of shared data
* present software and hardware solutions to the critical section problem
* examine classical process synchronization problems
* explore tools used to solve process synchronization problems

SECTION 5.1 - Background

* A process may be interrupted at any time. Some other process could execute arbitrary instructions next, before the first process is able to resume.
* This random interleaving of the actions of two processes can lead to incorrect usage of shared memory or resources.
* As an example, our text shows how the value of an integer variable can be corrupted if one process tries to increment it as another process tries to decrement it concurrently (or in parallel).
* This kind of situation is known as a *race condition*.
* The points made above illustrate the need for the OS to support process synchronization and coordination.

SECTION 5.2 - The Critical Section Problem

* We may need to designate critical sections in the code of groups of processes, where they access shared resources. We require that no two of them execute in their critical sections concurrently.
* How do we enforce this requirement? That is the critical section problem.
* Most solutions involve creating a protocol in which a process must get permission for exclusive access before entering a critical section, and must release its rights to the critical section after leaving it.
* One simple way to do this is to create a gatekeeper process that takes requests from the other processes, and tells them when they can take a turn executing their critical sections. There are situations when going through such a gatekeeper can be a bottleneck to performance.
* In this chapter we explore distributed solutions to the critical section problem, in which the processes behave symmetrically and cooperate as peers to synchronize.
* Typically we insert a section of *entry code* prior to each critical section (CS). A process executes entry code to gain permission to enter the CS.

Similarly, we insert a section of *exit code* after each CS, which processes use to release their rights to executed in the CS.

* We use the term *remainder section* to refer to the part of the process code that is not entry code, critical section, or exit code.
* **THIS IS VERY IMPORTANT:** Normally, we require that a solution to a critical section problem satisfy the following three requirements

(In the following, assume that $\{P_0, P_1, ..., P_{n-1}\}$ is a set of processes, and that each has a critical section that accesses some resource that they all share.

1. **Mutual Exclusion:** If one of the processes in the set, P_i, is executing in its CS, then none of the other processes in the set is executing in its CS.
2. **Progress:** If none of the processes is executing in its CS and some processes wish to enter their CSs, then only those processes that are executing in entry code or exit code can participate in deciding which process will enter its CS next, ***and*** that selection will not be postponed indefinitely.
3. **Bounded Waiting:** There must exist an a priori bound (a limit) on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted.

* We assume that each process is able to execute its instructions at some minimal rate, but there's no limit to how different the relative speed of processes could be.
* Often an OS kernel consists of multiple processes, which are subject to critical section problems (race conditions).
* Examples of shared kernel resources prone to race conditions
  + open file lists
  + memory allocation data
  + process lists
  + interrupt handling data
* On a uniprocessor, the OS can mask interrupts and refuse to relinquish the CPU until it has finished executing a critical section. This insures mutual exclusion. The is the idea of a ***non-preemptive kernel.***
* If multiple kernel threads are executing on a symmetric multiprocessor, then mutual exclusion cannot be assured merely by making the kernel non-preemptive.
* A ***preemptive kernel*** allows kernel processes/threads to be interrupted. Preemptive kernels are usually more responsive, and better at supporting real time computing.

Basic Set of Material to Cover in Lecture on Chapter Five of 9th Edition of Silberschatz. Process Synchronization

SECTION 5.3 - Peterson's Solution

* The structure of Peterson's solution is:

```
shared int turn ;
shared boolean flag[2] ;
void Ptrsn (int me; int you)
do
{
  flag[me]= true ;
  turn = you ;
  while (    flag[you]
          && turn you)
     /* do nothing */ ;
  CS(me) ;
  flag[me] = false ;
  RS(me) ;
} while(true);
```

* Peterson's solution solves the critical section problem for the case of two processes, assuming that the hardware implements int and boolean loads and stores atomically.
* What does ***atomically*** mean in the last sentence? It means that if two or more processes attempt to execute a load or store concurrently on an int or boolean, the **hardware** of the computer resolves the race condition - it picks an order for the operations to be performed, and performs them one at a time, each one in its entirety before the next is allowed to begin.
* Process 0 executes Ptrsn(0,1) and Process 1 executes Ptrsn(1,0).
* CS() and RS() are assumed to be functions that execute the critical sections and remainder sections of the processes, according to the values of the parameter given.
* Under the assumptions given, Peterson's solution satisfies all three of these requirements: mutual exclusion, progress, and bounded waiting.

SECTION 5.4 - Synchronization Hardware

* Modern operating systems often rely on special atomic hardware instructions to provide the support needed to implement solutions to critical section problems.
* **Examples:** 1) atomic test-and-set instruction, 2) atomic swap instruction, and 3) atomic compare-and-swap instruction
* Description of what a test-and-set instruction does:

```
bool test-and-set(bool *targ)
{
  bool rv = *targ ;
  *targ = true ;
  return rv;
}
```

* An atomic test-and-set must be implemented atomically in the hardware (the instruction set).
* When any two processes attempt to execute test-and-set on a parameter, the hardware serializes the operations - one process executes the operation

entirely, and then the other executes it entirely after the first has finished.

* How to provide mutual exclusion with test-and-set:

```
shared bool L=false ;
void beExclusive(int me)
{
  do
  {
    while (test-and-set(&L))
       /* do nothing */ ;
    CS(me) ;
    L=false ;
    RS(me) ;
  } while (true) ;
}
```

* Process #i executes beExclusive(i). Two or more processes can implement mutual exclusion this way.
* beExclusive does not provide progress or bounded waiting. However we can obtain them by augmenting beExclusive like this:
*

```
shared int n ;
shared bool waiting[n] ;
shared bool L=false ;

 /* Initialize all waiting[i]
    to false */
void beCS-solution(int me)
{
    /*local var, not shared*/
  int you;
  do
  {
    waiting[me]=true;
    while(waiting[me]
          && test-and-set(&L))
      /* do nothing */ ;
    waiting[me]=false;

    CS(me);

    you = (me+1)%n;
    while((you != me)
            &&(!waiting[you]))
      you=(you+1)%n;
    if (you==me) L=false ;
    else waiting[you]=false;

    RS(me) ;

  } while (true) ;
}
```

Basic Set of Material to Cover in Lecture on Chapter Five of 9th Edition of Silberschatz. Process Synchronization

SECTION 5.5 - Mutex Locks

* Code like the while-loop in beExclusive can be used as an acquire-lock() function to gain a lock, and something like the line L=false can be used to implement a release-lock() function.
* This is the idea of a mutex lock variable, or spin-lock.
* This kind of mutex uses busy-waiting, which can be a disadvantage. The code continually executes instructions in the CPU while waiting to acquire the lock. If the wait is long, then much CPU time may be wasted.
* If a process suspends itself while waiting to acquire a lock, then the CPU can be utilized for productive work. In particular, this may give the process holding the lock the opportunity to finish using it and release it sooner.
* One possible advantage of a spin-lock is that if the wait is short, then it does not require the delay of suspending and resuming the waiting process.
* It can be a good strategy to busy-wait for a lock on one CPU if the process holding the lock is executing on another CPU.

SECTION 5.6 - Semaphores

5.6.1 & 5.6.2 Semaphore Usage and Implementation

* The idea of a binary semaphore is roughly equivalent to a mutex variable.
* Counting semaphores are quite a bit different.
* For one thing, counting semaphores are designed to suspend waiting processes instead of using busy waiting.
* A counting semaphore has this kind of structure

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

* In other words, a semaphore is a kind of data object with two fields, an int value and a list of processes.
* The wait operation, which must be implemented atomically, does this:

```
wait(semaphore *S)
{
  S->value--;
  if(S->value<0)
  {
     put self in S->list;
     block();
  }
}
```

* The signal operation, which also must be implemented atomically, does this:

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

* block() suspends the process that invokes it, and wakeup(P) resumes the blocked process P.
* block() and wakeup() would normally be system calls.
* To help programmers assure progress and bounded waiting, the list of processes may be implemented as a FIFO queue. The semaphore data type we use in class projects is a counting semaphore with a FIFO queue for its list.
* Programmers use semaphores to solve critical section and other process synchronization problems.
* The solution of a critical section problem goes like this

```
shared semaphore S ;

wait(S);
CS(me);
signal(S);
```

* The problem of implementing semaphores atomically is itself a critical section problem. On a uniprocessor, one can make wait() and signal() system calls, and assure their atomicity by inhibiting interrupts. One can also use the technique illustrated by the code samples in section 5.4. This latter method is workable on a multiprocessor, as well as a uniprocessor.
* If the techniques of section 5.4 are employed, there will be some busy waiting, but this will happen only while one process waits for another process to complete a very short section of code (the amount of code in a wait or signal operation).

5.6.3 - Deadlock and Starvation

* Below we describe a deadlock scenario - a situation in which each process P in a group is waiting for one of the other processes to do something before P will do anything. Because every process in the group is waiting, none of them make any progress.

```
shared semaphores S,Q ;
  /* S & Q are counting
  semaphores with values
  initialized to 1. */
```

Code for P_0:

```
wait(S);
wait(Q);

.. other code ...

signal(S);
signal(Q);
```

Code for P_1:

```
wait(Q);
wait(S);

.. other code ...

signal(Q);
signal(S);
```

* Suppose that the execution of P_0 and P_1 is interleaved in the following way:

1) P_0 executes wait(S)
2) P_1 executes wait(Q)
3) P_0 executes wait(Q)
4) P_1 executes wait(S)

* In steps 3 and 4 P_0 and P_1 block on semaphores Q and S. Now both are suspended, each waiting for the other to signal on Q or S so that a call to wakeup() will allow them to proceed. They are destined to wait forever - nothing in the code provides a means to end their waiting.
* Deadlock causes ***infinite postponement*** - postponement that lasts forever.
* Another form of postponement - which is different - is called ***indefinite postponement***. Indefinite postponement is also called ***starvation***.
* Indefinite postponement is postponement for which there is no known upper bound to its length. It is like being imprisoned with an indeterminate sentence - your captors may or may not release you eventually, but they can keep you waiting as long as they want.
* On the other hand, infinite postponement is like a life sentence without possibility of parole. You know you are never going to be released.

5.6.4 - Priority Inversion

* The text describes a situation where three processes have three priorities L < M < H. P_H, with priority H, needs resource R, which is held by process P_L, running at priority L. Process P_M, running at priority M, gains the use of the CPU because it has higher priority than P_L, and because P_H is waiting for P_L to release R. Now P_M can take a long time to execute, thus preventing P_L from running to the point where it can release R. In effect P_H is being stalled by P_M, which has a lower priority than P_H. This is an example of priority inversion.

* One way to prevent priority inversion is to enforce priority-inheritance. In the example above, P_L would inherit the priority of P_H until P_L releases R, which would prevent the scheduler from giving preference to P_M over P_L.
* There's an inset in the text that explains how a case of priority inversion threatened the success of the Mars Pathfinder mission.

SECTION 5.7 - Classic Problems of Synchronization

* These are problems commonly used to test proposed process synchronization tools.

5.7.1 - The Bounded Buffer Problem

We can solve the bounded buffer problem by encapsulating the functionality of the counter in semaphores, as illustrated by the following code.

```
---------------------
#define BUFFER_SIZE 10
typedef struct
{
   /* here declare desired fields
     for the buffer item type */
} item ;
shared array int
buffer[BUFFER_SIZE];
 /*next position to add an item*/
shared int
in=0,
/* next position to remove an item */
out=0;
/* full.value == # full buffers */
shared semaphore
full(0),
/* empty.value == # empty buffers */
empty(BUFFER_SIZE) ;
---------------------
Producer's Code
do
{
   item nextp;
     /* produce an item in nextp */
   wait (empty) ;
   buffer[in]= nextp ;
   in =(in+1)%BUFFER_SIZE;
   signal(full) ;
} while (TRUE) ;
---------------------
Consumer's Code
do
{
   item nextc;
   wait (full) ;
   nextc = buffer[out] ;
   out=(out+1)%BUFFER_SIZE;
   signal(empty) ;
   /* consume nextc */
} while (TRUE) ;
---------------------
```

* This code uses the values in the semaphores to keep track of how many full and empty buffer slots exist. Also it delays the processes when necessary by blocking them.

5.7.2 - The Readers-Writers Problem

* The setup: a database is shared by a group of processes. Some are read only (readers). Some processes may write (writers). We have to maintain exclusive access for writers, but readers are allowed to access the database concurrently.
* The following pseudo-code describes a solution to the **first readers-writers problem**, which requires that no reader be kept waiting unless a writer has already obtained permission to access the database.

shared semaphore
rw_mutex=1, mutex=1;
shared int read_count=0;

Writer's Code:

```
do
{
  wait(rw_mutex);
    ... write to database ...
  signal(rw_mutex);
}while(true);
```

Reader's Code:

```
do
{
  wait(mutex);
  read_count++;
  if(read_count==1)
    wait(rw_mutex) ;
  signal(mutex);
    ... read from database ...
  wait(mutex);
  read_count--;
  if(read_count==0)
    signal(rw_mutex);
  signal(mutex);
}while(true);
```

* Basically, the code causes waiting writers to block on rw_mutex. A reader seeking access waits on rw_mutex only if no readers are currently reading or waiting. If more than one reader is waiting, one is blocked on rw_mutex, and the rest are blocked on mutex. Once there are readers accessing the data, any additional readers attempting access will be admitted immediately.
* This 'solution' allows writers to starve, even if the list in rw_mutex is implemented as a queue.
* Many OSs make read/write locks available. Such locks can be acquired either in read or write mode, and concurrent reading is supported.

5.7.3 - The Dining Philosophers Problem

* The problem can be stated in various ways. In one version, there are five 'philosopher' processes {P_0, P_1, P_2, P_3, P_4} and five disk drives, logically arranged in a circle, with a disk drive between each successive pair of processes. Every now and then, a process may need to copy data between the disk on its left and the disk on its right. The process needs to have exclusive access to the disks while doing the copy operation.
* An algorithm solving the problem must allow all the processes to operate without any of their copy operations experiencing indefinite (or infinite) postponement.
* One might naturally attempt to solve the problem like this:

```
shared semaphore drive[5];
   /* init values of all the
      drive[j] to 0 */
```

Code for P_i:

```
do
{
  wait(drive[i]);
  wait(drive[(i+1)%5]);
    ... do copy operation ...
  signal(drive[i]);
  signal(drive[(i+1)%5])
    ... do remainder section ...
}
```

* That code doesn't quite work. The problem is that deadlock is possible if all processes wait for their first semaphore at about the same time.
* There are many ways to solve the problem. One way is similar to the previous idea, except that each process is required to wait in for its semaphores in increasing numerical order.

SECTION 5.8 - Monitors

* Even with powerful tools like semaphores, process synchronization problems may be difficult to solve, and programmers may make mistakes even implementing well-understood solutions.
* For example, a programmer could place a call to signal() in a program that should be a call to wait().
* Such easily made coding errors can cause the software to operate very incorrectly, which can have very serious consequences. However such an error may go undetected until the timing of processes triggers a problem.
* For example, a computer program controlling the traffic lights in an intersection might have a bug that eventually causes a deadly collision between two cars, but it might take a long time before two cars enter the intersection at just the right times and

speeds to make the collision happen.
* Compilers can be created to perform some of the error-prone duties of programmers.

5.8.1 - Monitor Usage

* A monitor is an abstract data type. The programmer declares a monitor, which incorporates data and operations on the data.
* The compiler generates the code so that entrance into the monitor is atomic - no two processes can access data or execute functions within the monitor concurrently.
* The compiler may use such primitives as semaphores to implement a monitor.
* Thus the compiler takes over work that would otherwise be the responsibility of the programmer.
* Variables called conditions that are somewhat like binary semaphores are typically made available as part of monitors.

5.8.2 - Dining-Philosophers 'Solution' Using Monitors

* The authors present some code that uses a monitor to implement the dining philosophers 'situation'. The code presented is deadlock free, but it allows starvation - one or more philosophers could be delayed indefinitely from 'eating'.
* The idea of the solution is for a philosopher to wait until both chopsticks/disks are available and to acquire them both at the same time (atomically).

5.8.3 - Implementing a Monitor Using Semaphores

* Skip

5.8.4 - Resuming Processes Within a Monitor

* Skip

* Although the use of monitors can be helpful, errors in the code can still easily occur.

SECTION 5.9 - Synchronization Examples

5.9.1 - Synchronization in Windows

* Windows has a rich set of synchronization techniques and primitives, including the masking of interrupts, spin-locks, dispatcher objects, mutex locks, semaphores, events, timers, and critical section objects.

5.9.2 - Synchronization in Linux

* Linux has atomic integers, mutex locks, spin-locks, and semaphores. The latter two are available in plain and reader/writer lock versions.

* Linux also has the ability to enable/disable kernel preemption.

5.9.3 - Synchronization in Solaris

* Solaris has adaptive mutex locks, condition variables, semaphores, reader/writer locks, and turnstiles.
* A turnstile is a queue containing threads blocked on a lock.

5.9.4 - Pthreads Synchronization

* A Pthreads API provides mutex locks, condition variables, and reader/writer locks.
* Semaphores are not part of the Pthreads standard, although semaphores may be provided in systems that implement Pthreads.
* There are POSIX specifications for named and unnamed semaphores.
* (The semaphores we use in CS 3750 are a data type customized for use by the class, and are not part of the POSIX standard.)

SECTION 5.10 - Alternative Approaches

5.10.1 - Transactional Memory

* APIs support marking sections of code as requiring atomic execution.
* It is the responsibility of the compiler to generate code that treats this as a memory transaction that is either completed and committed or aborted and rolled back.
* Transactional memory may be implemented in software or hardware.

5.10.2 - OpenMP

* A programmer can mark an area of the program as a critical section and the compiler will generate code to enforce mutual exclusion.

5.10.3 - Functional Programming Languages

* Since functional programming languages are not focused on putting state variables through a series of changes, they can be useful for working around problems involving race conditions and deadlocks.
* Scala and Erlang are examples of languages used to write applications for parallel systems.