Synchronization
SGG: Chapter 6

I. Overview

A. Process model allows us an abstraction of a program (and the state to pause it)

B. Scheduling allows us concurrency and marshaled access to a scarce resource: CPU

C. Mechanisms for process communication and cooperation (shared-memory or message passing)

D. Threads allow us concurrency within a process and shared memory

1. With shared memory, there is potential for inconsistency

II. Critical Section

A. **Example**: Two processes accessing a reservation system, where the number of open seats is tracked by a counter

1. Process A: Releasing Seat
2. Process B: Reserving Seat
3. Counter at 10; Process A counter++ = 11; Process B counter-- = 9;
4. Whoever goes last wins. Either way, 11 or 9 is an incorrect value. This is a **race condition.**
5. Very common in multi-programming environments: allocating disk blocks or memory pages, writing to a file, network buffers, etc.
6. **Q:** What are some ways we can solve this?
   a) **Mutual Exclusion** (one at a time) on **Critical Sections** (the contention data/code)

B. Challenges:

1. How large can a critical section be?

2. How long can a process be in a critical section?
    a) What if they crash within a critical section?
3. How do we implement and who enforces entrance to critical sections?
4. **Example:** Beware naive implementations:

```
bool lock = FALSE
do {
   while (lock == TRUE);

   lock = TRUE;

   //CRITICAL SECTION

   lock = FALSE;

   }while(TRUE);
```

    a) Why not just have a single, binary value: lock?
        (1) Requires TWO operations: a read (test) and a write (set) that must be **atomic**
        (2) Interleaved lock access are deadly

C. Any solution must consider (and should satisfy):

1. **Mutual Exclusion**: Only one process can execute at a time.
2. **Progress**: Process *will* enter the critical section. And no process outside the critical region can block other processes.
    a) Only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next.
    b) E.g. A process cannot immediately re-enter the critical section if the other process want to.
3. **Bounded waiting**: In addition to guaranteeing entrance, limits on the amount of time in a critical section must be established.
4. **One more**: No assumptions on the speed or number of CPUs

D. Peterson's Solution: A software-based solution to critical sections (for two processes)

1. Two shared data items: `int turn; bool flag[2];`
    a) `turn` indicates whose turn it is: `turn == i` => P_i's turn
    b) if `flag[i] == true` => P_i is ready for the critical section

2. Algorithm (for process P_i):

```
do {
   flag[i] = TRUE;

   turn = j;

   while (flag[j] && turn == j);

   //CRITICAL SECTION

   flag[i] = FALSE;

   //REMAINDER

   }while(TRUE);
```

(1) To enter: P_i sets `flag[i] = true` and `turn = j`

(2) P_i demurs to P_j; if they are both ready `turn` will get set to?

(a) Who ever runs last sets the turn (to the other process)

3. **Q**: Did we meet our three requirements?

a) **Mutual exclusion**: Under what conditions can a process be in the critical section?

(1) P_i only enters if flag[j] is false or turn = i

(2) Conversely, if P_i and P_j are in the CS, then flag[i] == flag[j]; this implies both P_i and P_j terminated their whiles at the same time; BUT, in order for this to be true, turn must be both 0 and 1 (Contradiction)

b) **Progress**: Is there a condition where P_i can't enter the CS?

(1) There is no condition under which Pi cannot (eventually) enter the CS; either the other process will set turn == i or it is already set.

c) **Bounded waiting**: How long will P_i wait in the best/worst case?

(1) P_i will enter after at most one entry by P_j

4. Caveats: restricted to **two processes** (although a general solution exists)

III. Hardware Solutions

A. On a single processor: **disable** interrupts and preemption during critical sections

1. Not practical: single CPUs are rarer and rarer - disabling interrupts only affects one CPU at a time
2. Empowers (greedy) processes

B. Modern processors have support for locks through atomic instructions

1. TestAndSet()
2. Swap()
3. Both lock the memory bus (lighter weight than disabling interrupts)

C. TestAndSet: All operations in this function are **atomic**

```
bool TestAndSet(bool *lock){

   bool ret = *lock;

   *lock = TRUE;

   return ret;

}
```

```
lock = FALSE;

do{

   while(TestAndSet(&lock));

   // CRITICAL SECTION

   lock = FALSE;

}while(true)
```

D. Similar effect can be achieved with an atomic Swap() instruction. (good test question)

E. **Q**: Does the TestAndSet meet our requirements?

1. Mutual Exclusion: Yes
2. Bounded Wait/Progress? No. Pathological scheduling could result in a process waiting forever for the lock to be released.

V. Locks

A. Any solution to the critical section problem requires a **lock**

1. Peterson’s algorithm is an example (in software)
2. TestAndSet/Swap are hardware implementations

B. Solutions that continually test a lock exhibit **busy waiting**

a) A somewhat undesirable property: wastes CPU time

b) locks that use busy waiting are called **spin locks**

(1) Although, have the advantage of not causing a context switch, so if locks are fast, spin lock may be OK (we'll see this later)

c) Beyond wasting CPU, busy waiting can lead to incorrectness: **priority inversion problem**

2. **Example**: Suppose two processes P_h (high priority) & P_l (low priority), where P_h always gets scheduled before P_l.

a) P_h is ready and busy waiting

b) P_l is in its critical section, but never gets scheduled to complete

c) P_h loops forever

d) We'll see solution for detecting and avoiding deadlock in Chapter 7

VI. Semaphores

A. Semaphore is an integer variable used for synchronization

B. Accessed through two **atomic** operations: **wait()** and **signal()** (sometimes, P() "to test/try" and V() "to increment/raise" in Dutch)

```
wait(S){

   while(S <= 0);

   S--;

}
```

```
signal(S){

   S++;

}
```

C. Counting semaphore

1. S is initialized to the number of resources being protected by the semaphore

2. Decremented for each process that grabs a resource

D. **Binary semaphore**: A counting semaphore that goes to 1

1. Otherwise known as a **mutex lock** (mutual exclusion/mutex)

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

2. Can also be used for synchronizing execution
3. Example: sync = 0

```
//do something
signal(sync)
```

```
wait(sync)
//do something
```

4. pthreads support mutexes: pthread_mutex_*
   a) init; lock (lock or block); trylock (lock or fail (busywait)); destroy; unlock

E. **Blocking semaphores**: sleep() and wakeup()

```
wait(S){
   S->value--;
   if(S->value < 0){
      add process to S->list;
      sleep();
   }
}
```

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

1. Call to wait, and semaphore not available, process is blocked with sleep() (wait state) and placed on a waiting queue
2. Blocked processes are notified of an available semaphore by the wakeup() operation
   a) goes from waiting to ready
   b) Able to achieve bounded wait and progress with FIFO queue
3. A negative S value represents magnitude of waiting processes

F. How do we make semaphores **atomic**?

1. CPU can disable interrupts (or lock the data bus)

a) Semaphores can be a kernel controlled primitive, making this OK

2. For multiple CPUs: Use TSL instruction (test and set lock)
3. Still use busy wait
4. Note: Accessing a lock is a fast operation; so busy wait may be OK compared to busy waiting on a critical section of code

G. **Deadlocks** and **Starvation**

1. Synchronization primitives can be prone to deadlocks

a) Example: Two processes, two semaphores, each process waits on both semaphores in opposite order

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

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

2. Starvation can occur if we’re unfair in how we remove processes from the wait queue
3. We’ve already seen another example with priority inversion

H. Some Challenges (we’ll address later)

1. requesting a semaphore and forgetting to release it (or crashed while holding)
2. holding a semaphore for a long time without needing it

VII. Classic Synchronization Problems

A. Bounded-Buffer (aka a Producer-Consumer variant)

B. Dining-Philosophers (lots of variants) Always a good test question

1. Models process that want exclusive access to a limited number of resources
2. Description

a) N Philosophers P who think() and eat()

b) Food and N chopsticks

c) In order to eat() P must acquire() two chopsticks (in this example, left and right), each one at a time

d) In order to think(), both chopsticks must be release()’d

```
//a non-solution

void P(int i){

   while(1){

      think();

      acquire(chopstick[i]);

      acquire(chopstick[i+1 % N]);

      eat();

      release(chopstick[i]);

      release(chopstick[i+1 % N]);

  }

}
```

C. Q: The above is a non-solution: why?

   1. Deadlock

D. Q: What about trying for the second chopstick and putting down other chopstick?

   1. All pick up left, drop left, forever
   2. Starvation; livelock

E. Can we fix the above code?

   1. after think(): wait(mutex);
   2. after final release(): signal(mutex)
   3. Limitations?

      a) Performance: only one can eat at once

F. Q: Others?

```
#define N 5

#define LEFT (i+N-1)%N

#define RIGHT (i+1)%N

#define THINKING 0

#define HUNGRY 1

#define EATING 2

typedef int semaphore;

int state[N];

semaphore mutex = 1;

semaphore s[N];


void philosopher(int i){

   while(TRUE)

      think();

      take_forks(i);

      eat();

      put_forks(i);

   }

}
```

```
void take_forks(int i){

   down(&mutex); // enter CS

   state[i] = HUNGRY; //CS

   test(i);

   up(&mutex);

   down(&s[i]); //block if can’t eat

}

void put_fork(int i){

   down(&mutex);

   state[i] = THINKING;

   test(LEFT);

   test(RIGHT);

   up(&mutex);

}

void test(int i){

   if(state[i] == HUNGRY &&

    state[LEFT] != EATING &&

    state[RIGHT] != EATING){

      state[i] = EATING;

      up(&s[i]);

   }

}
```

G. General Solutions:

1. Exponential back off
2. Limit the number of philosophers who can eat at once
3. Only allow a philosopher to pick up both or neither forks (atomically)
4. Alternate even/odd philosophers

H. Another solution (above)

1. No deadlock or starvation, plus parallelism

2. Each philosopher has a state (thinking, hungry (need “forks”), or eating (has “forks”))
3. Philosopher may only move into EATING if neither neighbor is eating
4. Uses semaphores so hungry philosophers can block if “forks” are busy
    a) Unblocked by finishing neighbors

VIII. Monitors

A. Semaphores must be used with care

1. Subtle mistakes can lead to deadlock

B. **Monitors** were proposed by Tony Hoare (quicksort) as another primitive to make synchronization easier to program

C. Can think of monitors as a library with an API (abstract data type)

1. Processes share the library, but not internal data (directly)
2. This requires language specific understanding of a monitor (C doesn’t have them, natively)

D. Key idea: only one processes can be active within a monitor at any instant

1. Up to the compilers to ensure mutual exclusion on monitor procedures
    a) It can use other sync primitives to achieve this: e.g. a semaphore or mutex
2. We’re offloading synchronization correctness to the compiler, (hopefully) lessening the chance for error by the user

E. Monitor variables are private

F. A **condition** variable has two operations: wait() and signal()

1. A process that invokes `wait(x)` is blocked until another process invokes `signal(x)`
    a) signal will resume exactly one suspended process

b) Unlike a semaphore, signals are lost (stateless)

```
monitor ProducerConsumer

   conditional full, empty;

   int count;

   insert(int item){

      if (count == N) wait(full);

      insert_item(item);

      count++;

      if (count == 1) signal(empty);

   }

   int remove(){

      if(count == 0) wait(empty);

      remove = remove_item();

      count--;

      if(count == N-1) signal(full);

   }

}
```

G. In the above, all operations within the monitor are mutually exclusive

1. e.g. producer doesn’t need to worry about being interrupted before calling wait(full)
2. If buffer full, producer(s) are added to the “full” conditional’s wait queue
   a) Only a call to signal(full) (implies buffer no longer full), will a producer process be released from the queue

H. Resuming processes from conditional queue

1. FCFS
2. wait() can be modified to take a priority number
   a) priority number might be max time of resource use

IX. Barriers

A. Yet another primitive: but for a group of processes

1. Used for synchronizing processes into phases
2. e.g. applications that require many partial solutions to be complete before moving to next phase
   a) Scientific & parallel computing

X. Atomic Transactions & Consistency

A. Logging for consistency

1. Journaling file systems
   a) Committing data structures twice
2. Log-structured file systems
   a) Checkpoints and rollback

B. Eventual Consistency

1. Consistency over a wide area network
2. Transactions are committed and eventually make there way to all nodes
3. Timestamps and clocks can be used to sensibly order out-of-order transactions
4. So when is being eventually consistent good enough? Depends on design requirements.
5. Consider facebook or twitter updates: it’s unlikely you see tweets or status updates the second they are posted.
6. Does requires a different approach/mindset to programming
7. EC also affects durability; writing data is slow, delaying it makes it even slower. Crashed servers don’t received updates (dead men tell no tales); different NoSQL implementations have different levels of durability