628 Lecture Notes Week 6

Table of Contents

Announcements

- Midterm: Monday, March 6, 2017 (Week 8 of the course)
  - The midterm will include:
    - Lecture notes Week 1—Week 6
    - Labs 1–5.

Amdahl's Law (How much speed-up with multiple cores)

The extent to which a task can be sped up with multiple cores (or CPUs) depends on the fraction of the task that can run in parallel versus the fraction that must run sequentially (serially).

Gene Amdahl (IBM's chief computer architect in the 1960's) gave the following equation:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B+\frac{1}{n}(1-B)\right)} = \frac{1}{B+\frac{1}{n}(1-B)}$$

where: $S(n)$: Speed-up for *n* processors
B: the fraction of the algorithm that is serial.
T(1): Time for one core.
T(n): Time with *n* cores.

For example:

- If it is *all* serial (B = 1), then *S(n) = 1*
- If *all* can be run concurrently (B = 0), then *S(n) = n*
- If 20% is serial and there are 10 cores, then *S(10) = 3.5*
- If 10% is serial and there are 10 cores, then *S(10)* = 5.2
- If 10% is serial and there are 100 cores, then *S(10)* = 9.2

Mutual exclusion techniques

Disable interrupts

- Since a context switch (changing from one process/thread to another) can only happen due to an interrupt, disabling interrupts will avoid a process switch.
- Unfortunately, this is a *privileged instruction* that only the OS kernel can do.
- This is only a permissible policy for kernel code. **AND** it works if there is only one CPU.
- Alas, if there is more than one CPU (or core), it only works if all CPUs disable interrupts simultaneously...

Test and Set

- Any single machine language instruction is atomic.
- However, conditional branches often require more than one instruction, for example:

```
test memoryLocation
branchIfZero somewhere
```

- Suppose the memory location is zero and an interrupt and context switch occurs after the test instruction but before the branch instruction. Suppose further that the other thread or process make the "memoryLocation" non-zero. Oops!
- An *atomic* machine language instruction that allows you to **SET** a variable to a value **ONLY IF** it has some assumed value. (The "values" are often simple booleans.)

```
boolean test_and_set(boolean * flag) { //pseudo-C code
```

```
            boolean old = *flag;
            *flag = true;
            return old;
      }

      boolean some_lock = false;

      acquire(boolean * some_lock) {
            while(test_and_set(some_lock)) //spin until old was false
                  ;
      }

      release(some_lock) {
            lock = false;
      }

lock   fcb 0

acquire   tas lock   //imaginary test_and_set hcs12 instruction
          bne acquire

release   clr lock
```

- A similar atomic instruction is "compare and swap" which compares a memory location value to one operand and only changes its value to the second operand if the value and the first operand are the same.

Peterson's algorithm

- An algorithm that guarantees exclusive access to a critical region without disabling interrupts or using special machine language instructions (such as TestAndSet or CompareAndSwap).
- The algorithm is illustrated below assuming only 2 Threads. Each Thread tries to acquire a lock. Only one Thread can acquire the lock; it does its work and then releases the lock.
- We assume that interrupts (and hence switching from one Thread to another) can occur at any time.
- We also assume that the Thread IDs of the 2 Threads are 0 (for one Thread) and 1 (for the other).
- The following global variables are shared by both Threads:
  - boolean wants[2] = {false, false}; //Indicates that Thread i wants to acquire the lock
  - int turn = 0; //Indicates whose "turn" it is to acquire the lock.
- To acquire the lock Thread *i* does the following:

```
acquire:  wants[i] = true;  //Indicate Thread i wants the lock
          int other = i - 1;
```

```
turn = other;      //Give away i's turn to other Thread!
while(wants[other] && (turn != i)
    ;   //Spin lock until condition is false
```

- When the Thread acquires the lock and completes its work, it releases the lock with the following code:

```
release:  wants[i] = false;
```

- See also http://www.csee.wvu.edu/~jdm/classes/cs550/notes/tech/mutex/Peterson.html
- or Wikipedia: http://en.wikipedia.org/wiki/Peterson's_algorithm

Interprocess Communication

See also pages 40—43 in rymos notes (Note: there is an error on page 42)

Semaphores

Messages

Monitors

Signals (POSIX)

Pthread conditionals

Some standard problems

Producer-consumer

Using semaphores:

```
semaphore mutex = 1, empty =N, full = 0;
producer() {
     while(1) {
          produce()
          down(empty)
          down(mutex)
          insert_item()
          up(mutex)
          up(full)
     }
}

consumer() {
     while(1) {
          down(full)
          down(mutex)
          remove_item(
          up(mutex)
          up(empty)
     }
}
```

Using pthread

```
int buffer = 0;  //0 means empty buffer
pthread_mutex_t mutex
pthread_cont_t cond_cons, cond_prod

producer() {
     for(int i = 1; i < N; i++) {
          pthread_mutex_lock(&mutex)
          while(buffer != 0) {
               pthread_cond_wait(&cond_prod, &mutex)
          buffer = i
          pthread_cond_signal(&cond_cons)
          pthread_mutex_unlock(&mutex)
     }

consumer() {
     for(int i = 1; i < N; i++) {
          pthread_mutex_lock(&mutex)
          while(buffer == 0)
               pthread_cond_wait(&cond_cons, &mutex)
          buffer = 0;
          pthread(cond_signal(&cond_prod)
          pthread_mutex_unlock(&mutex)
```

```
      }
}
```

Using messages

```
#define N_BUFFER_SLOTS 10
producer() {
      while(1) {
            produce_item()
            receive(consumer, &msg)
            build_msg()
            send(consumer, &msg)
      }
}
consumer() {
      for(int i = 0; i < N_BUFFER_SLOTS; i++)
            send(producer, &msg);  /send N "empties"
      while(1) {
            receive(producer, &msg);
            get_item(&msg)
            send(producer, msg)
            consume_item()
      }
}
```

Readers-Writers

- Not covered before midterm.

Dining Philosopers

Banker's Algorithm (Deadlock Avoidance)

- The *Banker's Algorithm* allows the OS to avoid any possibility of deadlock when these conditions are met:
    - Each process (or thread) guarantees **before it starts** the maximum number of shared

resources that it will require before completion.

- Once a process has been allocated its maximum allotment of all the resources it needs, it will finish “in a reasonable amount of time”

- The algorithm is called “Banker's” because it allows a bank to grant “lines of credit” to its clients and will grant the request for credit only when:
  - Each client has been granted a line of credit which they will not exceed.
  - Each client is *guaranteed* that “at some time” they will be able to use their entire line of credit.
  - Each client will promptly repay their loan from their line of credit once they have withdrawn their maximum.
  - The Bank has limited resources (cash on hand) and will only grant a client's request for additional funds if and only if:
    - There is enough cash;
    - The client is not requesting something over their credit limit;
    - **AND** granting the request *guarantees* that there is “some sequence” of future requests whereby everyone can use their entire line of credit.
    - This last condition is the critical one: if true, the request can be granted and the bank is in a **safe** state; otherwise, granting the request would place the bank in an **unsafe** state.
    - The bank only grants requests that result in a “safe state”
  - Let's look at a simple example:
    - Suppose the bank has $15 in cash.
    - Three clients—Peter A, Peter B and Peter C—each have a $10 line of credit.
    - Suppose each requests and is granted $2 from their line of credit; the Bank still holds $9 and this is a **safe** condition because Peter A has $8 remaining in his line of credit so he can be satisfied. He will then pay back the $10 borrowed so the bank will have $11, enough to satisy Peter B or Peter C.
    - Suppose, however, that all clients request $1 more. If the bank grants all 3 requests, it would result in an **unsafe** condition; there would only be $6 left in the bank and each client would need $7 ot obtain their full line of credit. Consequently, the bank would not grant all 3 requests. (It would put one or two of them on a waiting list.)&, 5, 3)
    - The bank could, hwever, grant 2 of the requests—Peter A and Peter B—for example. It would retain $7 in cash and could fully satisfy either Peter A or Peter B.

Using the Banker's algorithm in an Operating System

- In general, there is more than one kind of resource (not just “cash”). In a computer system

memory or temporary disk space could be resources.

- Instead of clients, we have processes (or threads).
- Before starting, a process indicates the maximum quantity of each kind of resource it will need to complete. These numbers are the process's *maximum needs.*
- For example Process 1 may ultimately need (7, 5, 3) units of resources (A, B, C).
- The table below shows the usage and requirements for each process before any of them have been granted any resources by the OS.
- The bottom 2 rows summarize the total requirements for all processes and the last row shows how many resources in total are still available.

Process	**A**			**B**			**C**		
	Used	**Max**	**Needs**	**Used**	**Max**	**Needs**	**Used**	**Max**	**Needs**
P_0	0	7	7	0	5	5	0	3	3
P_1	0	3	3	0	2	2	0	2	2
P_2	0	9	9	0	0	0	0	2	2
P_3	0	2	2	0	2	2	0	2	2
P_4	0	4	4	0	3	3	0	3	3
Totals	**0**	**25**	**25**	**0**	**12**	**12**	**0**	**12**	**12**
OS avail	**10**			**5**			**7**		

Table 1: Initial state prior to any resource allocation

- After some time has elapsed the OS as granted various requests. For example P_0 has been granted 1 B resource, P_1 has been granted 2 A resources, P_2 has been granted 2 C resources.

Process	**A**			**B**			**C**		
	Used	**Max**	**Needs**	**Used**	**Max**	**Needs**	**Used**	**Max**	**Needs**
P_0	0	7	7	1	5	4	0	3	3
P_1	2	3	1	0	2	2	0	2	2
P_2	3	9	6	0	0	0	2	2	0
P_3	2	2	0	1	2	1	1	2	1
P_4	0	4	4	0	3	3	2	3	1
Totals	**7**	**25**	**18**	**2**	**12**	**10**	**5**	**12**	**7**
OS avail	**3**			**3**			**2**		

Table 2: Resources allocated, max and still needs after some allocations granted

- Is this safe?
- Yes. The sequence:
  - There are sufficient resources to satisfy all of P_1 requirements, so
  - P_1 exits leaving (A, B, C) total available (5, 3, 2)
  - P_3 can now finish leaving (A, B, C) total available (7, 4, 3)
  - P_0 can now finish leaving (A, B, C) total available as (7, 5, 3)
  - P_2 can now finish leaving (A, B, C) total available as (9, 5, 5)
  - Finally P_4 can now finish leaving (A, B, C) total available as (10, 5, 7)
- Suppose P1now requests (1, 0, 2). Would the resulting state be safe?
  - If granted, the allocation table would look like:

Process	A			B			C		
	Used	**Max**	**Needs**	**Used**	**Max**	**Needs**	**Used**	**Max**	**Needs**
P_0	0	7	7	1	5	4	0	3	3
P_1	3	3	0	0	2	2	2	2	0
P_2	3	9	6	0	0	0	2	2	0
P_3	2	2	0	1	2	1	1	2	1
P_4	0	4	4	0	3	3	2	3	1
Totals	**8**	**25**	**18**	**2**	**12**	**10**	**5**	**12**	**7**
OS avail	**2**			**3**			**0**		

- The resulting resource table is safe and all processes can finish with the same sequence as before: P_1 , P_3 , P_0 , P_2 , P_4

- Now suppose P_4 requests (3, 3, 0). Can the request be granted and, if so, would it result in a safe state?
  - The OS does not have 3 A resources; the request would be put onto a "waiting list" (i.e. the process would be Blocked) until the resources became available.
- Suppose P_0 requests (0, 2, 0). Can the request be granted and, if so, would it result in a safe state?
  - This time there are sufficient B resources to satisfy the request. ***If*** granted, the resource allocation table ***would*** become:

Process	A			B			C		
	Used	**Max**	**Needs**	**Used**	**Max**	**Needs**	**Used**	**Max**	**Needs**
P_0	0	7	7	3	5	2	0	3	3
P_1	3	3	0	0	2	2	2	2	0
P_2	3	9	6	0	0	0	2	2	0
P_3	2	2	0	1	2	1	1	2	1
P_4	0	4	4	0	3	3	2	3	1
Totals	**8**	**25**	**18**	**4**	**12**	**8**	**5**	**12**	**7**
OS avail	**2**			**1**			**0**		

  - This would result in an **unsafe** state.

- P_0 cannot have its B resources satisfied.
- P_1 cannot have its B resources satisfied.
- P_2 cannot have its A resources satisfied.
- P_3 cannot have its C resources satisfied.
- P_4 cannot have its A, B or C resources satisfied.