UM-Dearborn - CIS-450-002 Operating Systems

Student: Demetrius Johnson

Professor: Dr. Jinhua Guo

Homework 1

February 8, 2022

Due Date: February 18, 2022

CIS450/ECE478 Homework #1, Due Monday, Feb. 14, 2022

Q1. (10 points) How many new processes are created in the below program assuming calls to fork succeeds? Explain your answer.

```
int main(void)
{
   for (int i = 0; i < 4; i++) {
      pid_t pid = fork();
   }
}
```

Answer: There will be **15 new processes created** assuming all calls to fork() succeed. The solution is shown below: if you add up all highlighted areas that say "*n* new processes:", they will add up to 15. A call to fork() creates a copy of the parent and begins execution at the instruction position *after* the call to fork(), as I noted below; thus, each process will exit loop and the value of *i* will be incremented by 1, creating more children by iterating through the loop until *i* == 4, then it will return 0. Note, to save space in my solution below, I did not put return0 in every call to fork(), only in the call to fork() where *i*==3, such that it will exit the loop and increment to 4 and thus not create any new child processes. I did this to at least demonstrate and emphasize the point where no additional children would be created, and execution will be returned to the parent process.

*Note: if a parent enters a loop and initializes it, no reinitialization occurs by child processes; all data including initialized variables/loops are copied to the child process; execution begins in the child process ***after*** the fork() call.

- **P0 (starting process):**
  - **i = 0** at first loop entry; loop initialization occurs
  - loop iterates as i goes from [0 to 4) = 4 new processes:
  - P0/P1 (*all child processes follow this same logic*):
    - i = 0 → then starts at beginning of loop; i++ → i =1
    - [1 to 4) = 3 new processes:
    - P0/P1/P1:
      - i = 1
      - [2 to 4) = 2 new processes:
      - P0/P1/P1/P1:
        - i = 2
        - [3 to 4) = 1 new processes:
        - P0/P1/P1/P1/P1:
          - i = 3
          - (4 to 4) = 0 new processes
          - Return0
      - P0/P1/P1/P2:
        - i = 3
        - (4 to 4) = 0 new processes
        - Return0
    - P0/P1/P2:
      - i = 2
      - [3 to 4) = 1 new processes:
      - P0/P1/P2/P1:
        - i = 3
        - (4 to 4) = 0 new processes
        - Return0
    - P0/P1/P3:
      - i = 3
      - (4 to 4) = 0 new processes
      - Return0
  - P0/P2:
    - i = 1
    - [2 to 4) = 2 new processes:
    - P0/P2/P1:
      - i = 2
      - [3 to 4) = 1 new processes:
      - P0/P1/P1/P1:
        - i = 3
        - (4 to 4) = 0 new processes
        - Return0
    - P0/P2/P2:
      - i = 3
      - (4 to 4) = 0 new processes
      - Return0
  - P0/P3:
    - i = 2
    - [3 to 4) = 1 new processes:
    - P0/P3/P1:
      - i = 3
      - (4 to 4) = 0 new processes
      - Return0
  - P0/P4:
    - i = 3
    - (4 to 4) = 0 new processes
    - Return0

Q2. (15 points) Considering the following code

```
int y = 5;  x = 1;
void foo () {
   y = y + 1;
   x = y;
}
void bar () {
   y = y * 2;
}
main () {
   Thread t1 = createThread(foo);
   Thread t2 = createThread(bar);
   WaitForAllDone();
   cout << x  << y <<endl;
}
```

Note that function WaitForAllDone blocks the main thread until both threads it creates are done. Assume that memory load and memory store are atomic, but add and time are not atomic. Note,

y = y + 1, consists of three assembly instructions :
Load R1, y
Add R1, 1, R1
Store R1, y

x = y, consists of two assembly instructions :
Load R1, y
Store R1, x

y = y * 2, consists of three assembly instructions :
Load R1, y
Mul R1, 2, R1
Store R1, y

Registers not shared between the threads.

(a) What is the maximum number of threads that are ever alive (this includes those on the ready or waiting lists, or running) in this program?

If you do not include the main() function, the maximum number of threads that are ever alive in this program is **3**: the foo() and bar() functions, as well as the WaitForAllDone() function. While foo() and bar() take turns running, they also have to share the processor with the WaitForAllDone() function, which causes the main() function to keep being blocked and go to wait state. Thus, all at one time, all processes are alive. If you include the main() function, then the total number of threads alive at once is **4**.

(b)Give all possible outputs of this program. Explain your answer and be specific.

The cout << line will only execute after both the t1 and t2 threads exit as a result of the WaitForAllDone() function. The variables x = 1 and y = 5 are global variables and thus will always start at the given initialized values no matter how many time the main() program runs. Also note: the statement x = y (as highlighted below) is the only line that can never be the

first line to begin execution and affect the value of x or y, as it will always be *after* the second execution statement in foo(). Given that information, and the fact that *load* and *store* are the only atomic functions (and registers are not shared between threads, i.e., R1 for each thread is independent of R1 from all other threads), we can derive all possible outputs of this program. Note, some statements (such as *load* and *store*) are condensed/taken out for simplicity and saving space; there are more combinations using load and store that will generate the same outcome, but the total possible distinct outcomes are much less, and so we want to focus on deriving that information:

- Scenario 1:
  a. foo::R1=y+1 //R1=6, y=5, x=1
  b. foo::Store R1, y //R1=6, y=6, x=1
  c. foo::x=y //R1=6, y=6, x=6
  d. bar::*R1=y*2 //*R1=12, y=6, x=6
  e. bar::Store *R1, y //*R1=12, y=12, x=6
  f. main::cout(x,y) // final output == **x is 6, and y is 12.**
- Scenario 2:
  a. foo::R1=y+1 //R1=6, y=5, x=1
  b. bar::*R1=y*2 //*R1=10, y=5, x=1
  c. foo::Store R1, y //R1=6, y=6, x=1
  d. bar::Store *R1, y //*R1=10, y=10, x=1
  e. foo::x=y //R1=10, y=10, x=10
  f. main::cout(x,y) // final output == **x is 10, and y is 10.**
- Scenario 3:
  a. foo::R1=y+1 //R1=6, y=5, x=1
  b. bar::*R1=y*2 //*R1=10, y=5, x=1
  c. foo::Store R1, y //R1=6, y=6, x=1
  d. foo::x=y //R1=6, y=6, x=6
  e. bar::Store *R1, y //*R1=10, y=10, x=6
  f. main::cout(x,y) // final output == **x is 6, and y is 10.**
- Scenario 4:
  a. bar::*R1=y*2 //*R1=10, y=5, x=1
  b. bar::Store *R1, y //*R1=10, y=10, x=1
  c. foo::R1=y+1 //R1=11, y=10, x=1
  d. foo::Store R1, y //R1=11, y=11, x=1
  e. foo::x=y //R1=11, y=11, x=11
  f. main::cout(x,y) // final output == **x is 11, and y is 11.**
- Scenario 5:
  a. bar::*R1=y*2 //*R1=10, y=5, x=1
  b. foo::R1=y+1 //R1=6, y=5, x=1
  c. foo::Store R1, y //R1=6, y=6, x=1
  d. foo::x=y //R1=6, y=6, x=6
  e. bar::Store *R1, y //*R1=10, y=10, x=6
  f. main::cout(x,y) // final output == **x is 10, and y is 6.**
- Scenario 6:
  a. bar::*R1=y*2 //*R1=10, y=5, x=1
  b. foo::R1=y+1 //R1=6, y=5, x=1
  c. bar::Store *R1, y //*R1=10, y=10, x=1
  d. foo::Store R1, y //R1=6, y=6, x=1
  e. foo::x=y //R1=6, y=6, x=6
  f. main::cout(x,y) // final output == **x is 6, and y is 6.**

Q3.(10 points) Consider the following pseudo-code routines to provide a solution to the critical section problem between two threads in a uniprocessor system. In addition to the two disadvantages of "busy waiting" and "good for only two threads", there is a major flaw in the routines. Explain this major flaw:

```
Initially, turn == 0

entry (id) {
     while (turn !=id);  /* if not my turn, spin */
}

exit(id) {
     turn = 1 - id;      /* other thread's turn*/
}

Note, the id is the thread id (id == 0 or 1).
```

- Above, we have a two-thread critical section problem solution attempt.
- Some of the limitations/problems with the above solution are as follows:
  - Threads 0 and 1 must always *alternate* at trying to become materialized.
  - For example, when thread 0 finishes, the above code sets it so that thread 1 can enter. But what if a new thread 0 needs to enter again to be executed after the other thread 0 finished? It couldn't; it has to wait for a thread 1 to be processed; and vice versa applies for thread 1s, thus, **consecutive** threads of the *same type* is not allowed with the above solution.
  - Also, if id is set initially to 0, if a thread 0 does never enters, or if there is a long period of time before one is called to enter, then no initial thread 1 will be able to enter for the same reasons as stated above, and there will be an unnecessary wait time. The same can happen even after threads alternate for a period; if the last thread to finish exits, and then another thread of its type comes along, it may have to wait a long time until the alternate thread is called and enters and exits so that it can get a turn to enter the CS.
  - So, waiting is **not bounded,** and **Absence of Unnecessary Delay** are thus not satisfied for the four main principles that CS code solution should contain (but this example does have the mutually exclusive and absence of deadlock characteristics).

Q4.(10 points) Solve the critical section problem using the atomic increment instruction. Its definition is:

```
int Inc(int &x): {
    x = x + 1;
    return x;
}
```

This instruction executes atomically. Note also that x is a reference parameter. You are to provide the **entry()** and **exit()** routines, as in the Test-and-Set example in class. *Note, ignore the possible integer overflow.*

Notes on my solution below: I set x to -1 in order to allow the **atomic** Inc() function to set x and the spin variable to 0, which will allow the while loop to be skipped and a thread will be able to enter the critical section of code. Whenever there is already a thread in the CS, the thread does enter the loop because Inc() returns a value to spin that is >0; thus, it will loop until the thread currently in the CS exits() and resets x to -1; upon doing this, when we reset the value of spin in the next while loop, it will acquire entry because spin==0 (unless a context switch occurs and another thread is allowed to call Inc() first).

- The issue with this solution is **busy-waiting**: although it can handle any number of threads, is that it will constantly be in the CPU ready queue, so each time a thread is loaded into CPU that is spinning, the CPU will have to waste many cycles loading and executing the Inc() function, and then moving on to the next thread if the thread cannot enter the critical section. This waste of CPU resources is not the only issue; some threads may keep getting skipped and have **unbounded wait time**, due to the issue described above regarding threads acquiring the next 0 value by being loaded into CPU and calling Inc() first. Also, this solution does not provide any priority to higher-priority threads.

```
int x = -1  //x is a global var. initialized to -1

entry (){ //try to gain entry into CS code

      int spin;      //use this to check status and loop
      spin = Inc(x)  //Inc() is atomic; spin && x = x++

      while (spin){  //if spin == 0; can enter CS.
                       //otherwise, spin in this loop
           spin = Inc(x)} // spin && x = x++

                    }

*{...          //CRITICAL SECTION CODE HERE//        ...}*

     exit(){ //time to exit CS and allow other threads entry

          x = -1; //reset to -1; allow other threads entry

           }
```

Q5.(15 points) The local Laundromat has just entered the computer age. As each customer enters, he or she puts coins into slot at one of two stations and type in the number of washing machines he/she will need. The stations are connected to a central computer that automatically assigns available machines and outputs tokens that identify the machines to be used. The customer puts laundry into the machines and inserts each token into the machine indicated on the token. When a machine finishes its cycle, it informs the computer that it is available again. The computer maintains a Boolean array available[NMACHINES] to represent if corresponding machines are available (NMACHINES is a constant indicating how many machines there are in the Laundromat), and a semaphore nfree that indicates how many machines are available. The code to allocate and release machines is as follows.

The *available* array is initialized to all true, and *nfree* is initialized to NMACHINES.

```
int allocate() /*Returns index of available machine */
{
     wait(nfree); /*Wait until a machine is available */
     for (int i=0; i < NMACHINES; i++) {
          if (available[i]) {
               available[i] = FALSE;
               return i;
          }
     }
}
void release (int machine) /* release machine */
{
     available[machine] = TRUE;
     signal(nfree);
}
```

(a) It seems that if two people make requests at the two stations at the same time, they will occasionally be assigned the same machine. This has resulted in several brawls in the Laundromat, and you have been called in by the owner to fix the problem. Assume that one thread handles each customer station. Explain how the same washing machine can be assigned to two different customers.

One washing machine can be assigned to two different customers because of the following:

- When nfree >= 2, then two separate threads can exit the wait() function (via context switching) and move on to the for-loop and search for one of the available machines. It is this search part that is a **critical section** of code that can create indeterminate and overlapping token outputs; for example, one thread could select the first available machine in the array, and then a context switch can occur before that selected machine's availability is set to false, thus, allowing the next thread loaded into CPU to search the machine array and acquire the same machine. So, we have a *race condition* when two or

more threads are searching for available machines after two or more machines are available. There are no issues with the release function (at least as far I can tell).

(b)Modify the code to eliminate the problem, using semaphore only.

- The *available* array is initialized to all true, and *nfree* is initialized to NMACHINES.
- Now, to solve the CS problem during the search in the for-loop, let's add another semaphore. This needs only to provide mutual exclusion: we will use a binary semaphore.
  - Semaphore search = 1; We initialize it to 1 so that only one thread is allowed to search at a time; all other threads will be blocked and have to wait a short time until the current thread finishes its search and finds a machine.
  - This will solve the problem, and no more brawls should brake out as every customer will now always get a unique available token.
  - Note, we need to place the signal of the search semaphore *before* we return i, otherwise, no other threads will be able to enter the CS search code, and thus we will have a deadlock.

```
#DEFINE NMACHINES = (some number of machines);
bool available[NMACHINES] = (set all to true);
semaphore nfree = NMACHINES;
semaphore search = 1; //use this to make searching mutually exclusive

int allocate() //Returns index of available machine
     {
          wait(nfree);  //Wait until a machine is available
          wait(search); //Make sure no other thread is searching

          for (int i=0; i < NMACHINES; i++) {
               if (available[i]) {
                    available[i] = FALSE;
                    signal(search); //Allow another thread to search
                    return i;
               }
          }
     }
//No changes needed for the release function
     void release (int machine) // release machine
     {
          available[machine] = TRUE;
          signal(nfree);
     }
```

Q6.(20 points) Concurrent Sums with Monitors

```
int a, b, c, d, e;

cin >> a;
cin >> b;
c = b * 2;
d = b + 1;
e = d – a;
cout << a << e;
```

Your goal is to rewrite this program as a collection of threads, one for each line of code. (Each thread should call a monitor function that, possibly among other things, executes its line of code.) You must use monitors to retain the original sequential semantics of the code (i.e., the output must be identical to the same program where all statements are executed by the same thread). However, you must also exploit the maximum amount of concurrency; when two statements can safely execute concurrently, you must let them do so.

- Note: cin>> operation must be done in the correct sequence; otherwise, a and b assignment values from the original program could be swapped, thus giving a different result than the original program given above, and it could cause indeterminate results.
- Keep in mind, the condition variable in a monitor acts as a binary semaphore initialized to 0, and contains its own separate wait queue for when it sends threads to sleep via cond.wait().
- Note: multiple threads could be waiting for a signal, so at those locations I used a broadcast signal.
- I simply took advantage of the conditional variable function to tell when a given function could execute its operations.

```
Monitor currentCompute{

int a, b, c, d, e;
bool aAvail, bAvail, cAvail, dAvail, eAvail = false;
condition aReady, bReady, cReady, dReady, eReady;

	readA(){

		cin >> a;
		aAvail = true;
		aReady.broadcast();
	}
	readB(){

		if(!aAvail)
			aAvail.wait();
		cin >> b;
		bAvail = true;
		bReady.broadcast();
	}
	mulC(){

		if(!bAvail)
			bAvail.wait();
		c = b*2;
		cAvail = true;
		cReady.signal();
	}
	setD(){

		if(!bAvail)
			bReady.wait();
		if(!cAvail)
			cReady.wait();
		d = b+1;
		dAvail = true;
		dReady.signal();

	}
	setE(){

		if(!dAvail)
			dReady.wait();
		e = d - a;
		eAvail = true;
		eReady.signal();

	}
	output_AE(){

		if(!aAvail)
			aReady.wait();
		cout << a;
		if(!eAvail)
			eReady.wait();
		cout << e;
	}
}
```

Q7.(20 points) A Saving Account Problem

A saving account is shared by several people (threads). Each person may deposit and withdraw funds from the account. The current balance in the account is the sum of all deposits to date minuses the sum of all withdraws to date. The balance must never become negative. A deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds.

(a) Develop a monitor to solve this problem. The monitor should have two procedures: deposit(amount) and withdraw(amount). Assume the arguments to deposit and withdraw are positive.

- I feel pretty confident that for parts a and b by solutions are correct.
- Note: I kept in mind that when a thread leaves a condition queue, it goes back to the monitor entry queue; which both queues are by default FIFO. So for part b, the solution was to make sure a thread couldn't wake up and change it's priority until it was signaled to do so, and all the threads will be broadcasted to do so; thus they will – in order – decrement their priority and no withdraw can skip ahead of another withdraw thread.

```
Monitor savingsAcct{

int currentBal = (current balance to date);
condition suffcientBal;

	deposit(int x){

		currentBal += x;	//add x to current balance
		suffBal.broadcast();	//try all withdraw threads
	}



	withdraw(int x){

		while(currentBal < x )
			sufficientBal.wait(); //block withdraw thread
		currentBal -= x;		//balance is now sufficient for withdrawal
	}
}
```

(b) Modify your answer to (a) so that withdraws are serviced FCFS (first come first serve). For example, suppose the current balance is $200, and one customer is waiting to withdraw $300. If another customer arrives, he must wait, even if he wants to withdraw at most $200. Assume the waiting queue is FIFO.

```
Monitor FCFS_savingsAcct{

int currentBal = (current balance to date);
numWait_withdraws = -1; //priority 0 will be first priority
condition suffcientBal, changePriority;

	deposit(int x){

		currentBal += x; //add x to current balance
		sufficientBal.signal();
	}



	withdraw(int x){

		numWait_withdraws++;
		int priorityNum = numWait_withdraws; //assign a priority val; ensure no
	threads skip ahead
		while(priorityNum > 0){

			changePriority.wait();	//wait for signal to decrement priority
			priorityNum--;		//move closer to front of queue
		}

		while(currentBal < x)
			sufficientBal.wait();

		currentBal -= x;		//subtract x from current balance
		numWait_withdraws--;		//decrement queue
		changePriority.broadcast(); //all threads move to entry monitor queue; will
						//decrement priority upon waking up
	}
}
```