Question No.1

What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer No.1

Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

When a task cannot continue until an event occurs and so repeatedly polls to see if the event has occurred, it is said to be busy waiting. The key idea is that the task continues to consume processor time by repeatedly checking the condition. Examples of busy waiting include software synchronization algorithms. such as the bakery algorithm, and spin locks using hardware synchronization primitives such as test and set.

The alternative to busy waiting is blocked waiting (also known as sleeping waiting), where a task sleeps until an event occurs. For blocked waiting to work, there must be some external agent that can wake up the task when the event has (or may have) occurred.

Processes also wait when they are ready to run(not waiting for any event) but the processor is busy executing other tasks.

Busy waiting cannot be avoided altogether. Some events cannot trigger a wakeup; for example, on Unix a process cannot "sleep until a file is modified," because the operating system does not provide any mechanism to automatically wake up the process when the event occurs; some amount of repeated polling is required. (Windows NT, on the other hand, does allow this kind of wakeup.)

Also, blocking synchronization primitives such as semaphores need to use a lower- level synchronization technique in their implementation. On a uniprocessor machine, techniques such as disabling interrupts can enforce a critical section, but on multiprocessor machines, disabling interrupts is inadequate and some form of busy-waiting synchronization must be used.

Generally, busy waiting is mostly avoidable on uniprocessor machines, but is mostly unavoidable on multiprocessor machines. However, even on multiprocessor machines busy-waiting is best used for very short waits and limited to operating system code.

Question No.2

Illustrate how a binary semaphore can be used to implement mutual exclusion among *n* processes

Answer No.2

The *n* processes share a semaphore, mutex, initialized to 1. Each process *Pi* is organized as follows:

```
do {
wait(mutex);
/* critical section */
signal(mutex);
/* remainder section */
} while (true);
```

Question No.3

Describe how the compare_and_swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

Answer No.3

Compare-and-swap CAS is an atomic operation. This means that comparison and swap it performs are executed atomically. CAS needs three arguments: two "old" values (lets call them A and B here) and new value written in B (lets call it C). I will clarify with an example how CAS works.

Let A be a variable and that variable has some value, i.e. A 5. A programmer programmed that A should be multiplied by 10. This is what CAS does:

1. B = A, where B is a new variable
2. result = CAS(A,B,A*10)

Variable A is global. More than one process and therefore more than one thread can see variable A. If process $\mathbf{P_1}$ wants to multiply variable A by 10 using atomic operation CAS it first makes a local copy of A (in our case this is called B). It then atomically compares A and B, and if they are equal, A will be replaced by 10A. If they are not equal $\mathbf{P_1}$ will re-read value of A into B, and execute CAS instruction again. You can see from the explanation before that variable A could be changed by other process before $\mathbf{P_1}$ executes CAS. If that were to occur, process $\mathbf{P_1}$ would re-read value of A.

So how could one use this to implement a mutex? A variable with some special number (-1 because no processes use that pid) could indicate that mutex is free. If it is claimed, use CAS to write own pid into that variable. After you are done, write -1. in it.

Reference: https://answer.ya.guru/questions/6893543-514-describe-how-the-compare-and-swap-instruction-can-be-used-to.html

Question No.4

Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test_and_set() instruction. The solution should exhibit minimal busy waiting.

Answer No.4

Recall that test_and_set() operation writes 1 to a memory location and returns old value from that memory location. Obviously, anything can be written on that memory location, but only 1 can be written by test_and_set().

Our semaphore needs 4 variables: Two booleans which will act like mutex (l will explain why two), one counter and a queue. Semaphore counter and queue are self explanatory. When user wishes to call acquire() it would have to test_and_set() first boolean variable. This one represents a internal mutex (initially false) which guards the counter. Then, acquire() reduces the count by one, and signals the queue that thread in it can access the resource. It must also set first boolean variable to false. If acquire() can't reduce the count because it is already 0, it needs to wait until other thread calls release(). So, acquire() sets first boolean to false again, and waits by using test_and_set() on a second boolean which acts as a mutex for queue only. If semaphore only had one boolean variable a deadlock could occur.

release() function would perform similarly: Use test_and_set() on first boolean variable to enter critical region, increase the count on semaphore. If it is greater than 0, set false to second variable, set false to first variable and leave.

Question No.5

A semaphore puts a thread to sleep:

(a) if it tries to decrement the semaphore's value below 0.
(b) if it increments the semaphore's value above 0.
(c) until another thread issues a notify on the semaphore.
(d) until the semaphore's value reaches a specific number.

Answer No.5

Solution :

(a) if it tries to decrement the semaphore's value below 0.

Question No.6

What is the difference between Mesa and Hoare scheduling for monitors?

Answer No.6

For Mesa scheduling, the signaler keeps the lock and CPU, while the signaled thread is simply put on the ready queue and will run at a later time. Further, a programmer with Mesa scheduled monitors must recheck the condition after being awoken from a Wait() operation [i.e. they need a while loop around the execution of Wait(). For Hoare scheduling, the signaler gives the lock and CPU to the signaled thread which begins running until it releases the lock, at which point the signaler regains the lock and CPU. A programmer with Hoare scheduled monitors does not need to recheck the condition after being awoken, since they know that the code after the Wait() is executed immediately after the Signal() [i.e. they do not need a whileloop around the execution of Wait()].

Question No.7

The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

boolean flag[2]; /* initially false */
int turn;

The structure of process Pi (i == 0 or 1) is shown in Figure 1. The other process is Pj(j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical section problem.

```
do {
flag[i] = true;
while (flag[j]) {
        if (turn == j) {
                flag[i] = false;
                while (turn == j); /* do nothing */
        flag[i] = true;
          }
}
/* critical section */
turn = j;
flag[i] = false;
          /* remainder section */
} while (true);
```

Figure 1 The structure of process *Pi* in Dekker's algorithm

Answer No.7

This algorithm satisfies the three conditions of mutual exclusion.

(1) Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.
(2) Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting turn to the other process upon exiting.

(3) Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

Question No.8

Recall that semaphores can be used to implement mutual exclusion or thread scheduling dependencies. Show in pseudocode how a semaphore can be used to implement the join operation on a thread. Be sure to indicate the initial value of the semaphore. What would be the fill in the program at LINE A, LINE B and LINE C?

```
Thread::Thread() {
…
        if (joinable) {
         _____ //Line A：Your code goes here
         }
…
}
void Thread::Join() {
        ASSERT(joinable);
         _____ ///LineB：Your code goes here
         delete this;
}
void Thread::Finish() {
        kernel->interrupt->SetLevel(IntOff);
        _____ ///Line C：Your code goes here
if (joinable)
          Sleep(FALSE);
else

          Sleep(TRUE);
}
```

Answer No.8

First missing statement is "sema = 0" / must initialize the semaphore correctly
Second missing statement) is "sema.P() / sema.wait()"
Thirds missing statement is "sema.V() / sema.signal()"

Question No.9

The readers-writers problem relates to an object such as a file that is shared between multiple threads. Some of these threads are readers i.e. they only want to read the data from the object and some of the threads are writers i.e. they want to write into the object. Try to use Pthread semaphore API to implement the reader-writer problem.

Answer No.9

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.
This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows:

Reader Process

The code that defines the reader process is given below:

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);
.. //READ THE OBJECT
```

```
.wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

In the above code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

The variable rc denotes the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read operation is done, rc is decremented. When re becomes 0, signal operation is used on wrt. So a writer can access the object now.

Writer Process

The code that defines the writer process is given below:

```
wait(wrt);
……
 //WRITE INTO THE OBJECT
….
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on wrt. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on wrt.