Operating System

Kamruzzaman Asif

All Stuff: https://www.os-book.com/OS10/index.html

Book Solution Page:
https://codex.cs.yale.edu/avi/os-book/OS10/practice-exercises/index-solu.html

Chap 8: https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/index.html for synchronizaiton problem practice
TA: https://github.com/dpulsifer/Sleeping-TA-Problem

Chapter wise Solution to the Exercises:

chap-2

2.9
One class of services provided by an operating system is to enforce protection between different processes running concurrently in the system.

The second class of services provided by an operating system is to provide new functionality that is not supported directly by the underlying hardware.

2.10 Describe three general methods for passing parameters to the operating system.
a. Pass parameters in registers
b. Registers pass starting addresses of blocks of parameters
c. Parameters can be placed, or pushed, onto the stack by the program, and popped off the stack by the operating system.

2.15 What are the two models of inter process communication? What are the strengths and weaknesses of the two approaches?
Ans: The two models of inter process communication are message passing model and the shared-memory model.

2.19 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?
Ans: Benefits typically include the following (a) adding a new service does not require modifying the kernel, (b) it is more
secure as more operations are done in user mode than in kernel mode, and (c) a simpler kernel design and functionality
typically results in a more reliable operating system. User programs and system services interact in a microkernel
architecture by using inter process communication mechanisms such as messaging. These messages are conveyed by the
operating system. The primary disadvantage of the microkernel architecture are the overheads associated with inter
process communication and the frequent use of the operating system's messaging functions in order to
enable the user process and the system service to interact with each other.

2.20 What are the advantages of using loadable kernel modules?
Ans: It is difficult to predict what features an operating system will need when it is being designed. The advantage of
Using loadable kernel modules is that functionality can be added to and removed from the kernel while it is running.
There is no need to either recompile or reboot the kernel.

chap-3

3.17 What are the benefits and the disadvantages of each of the following?
Consider both the system level and the programmer level.
a. Synchronous and asynchronous communication
ANS:

> A benefit of synchronous communication is that it allows a rendezvous between the sender and receiver. A disadvantage of a blocking send is that a rendezvous may not be required and the message could be delivered asynchronously. As a result, message-passing systems often provide both forms of synchronization.

答案可參考第八版課本 P.122 , 第七版 P.99

b. Automatic and explicit buffering

ANS:

Automatic buffering provides a queue with indefinite length, thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications on how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely that memory will be wasted with explicit buffering.

答案可參考第八版課本 P.122~123 , 第七版 P.99

c. Send by copy and send by reference

ANS:

Send by copy does not allow the receiver to alter the state of the parameter; send by reference does allow it. A benefit of send by reference is that it allows the programmer to write a distributed version of a centralized application. Java's RMI provides both; however, passing a parameter by reference requires declaring the parameter as a remote object as well

d. Fixed-sized and variable-sized messages

ANS:

The implications of this are mostly related to buffering issues; with fixed-size messages, a buffer with a specific size can hold a known number of messages. The number of variable-sized messages that can be held by such a buffer is unknown. Consider how Windows 2000 handles this situation: with fixed-sized messages (anything < 256 bytes), the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages (i.e. variable-sized messages) use shared memory to pass the message.

3.13 Using the program in Figure 3.29, identify the values of pid at lines A, B, C and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main ()
{
 /* fork a child process */
 pid = fork();
 if ( pid < 0 ) { /*error occurred
 fprintf(stderr, "Fork Failed");
```

```
return 1;
}
else if (pid ==0) { /* child process */
pid1 = getpid();
printf("child: pid = %d" ,pid); /* A */
printf("child: pid1 = %d" ,pid1); /* B */
}
else { /* parent process */
pid1 = getpid();
printf("parent: pid = %d" ,pid); /* C */
printf("parent: pid1 = %d" ,pid1); /* D */
wait(NULL);
}
return 0;
}
```

Figure3.29 what are the pid values?

ANS:

A = 0, B = 2603, C = 2603, D = 2600

3.12

If the fork works but the execlp doesn't, *that's* when that line will be printed, and only in the child process.

3.16

LINE X

CHILD: 0 CHILD: 0 CHILD: 0 CHILD: 0 CHILD: 0

LINE Y

PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

Explanation: fork() system call will fork child Process and its return type is 0 for child process, Child will modify the array values all to 0 i.e 1-1 = 0 , 2-2 = 0, 3-3 = 0, 4-4 = 0 etc

Hence LineX produces Output as CHILD: 0 CHILD: 0 CHILD: 0 CHILD: 0 CHILD: 0

PARENT will execute else BLOCK

PARENT DOES NOT Modify the array ,

Hence LineY produces Output as PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

& some question ans: https://www.cram.com/flashcards/os340-chapter-3-5189174

chap-4

a) Provide two programming examples in which multithreading provides better performance than a single-threaded solution.
b) Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.
c) Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

1] i) A matrix multiplication where different parts of the matrix may be worked on in parallel.

ii) A web server that services each request in a separate thread.

2] i) A C shell program must closely monitor its own working space such as current working directory or open files etc.

ii) A sequential program that calculates an individual tax return.

3] A multithreaded solution using multiple user level threads does not achieve better performance on a multiprocessor system than on a single processor system.The operating system will not scheule different threads of process on separate processors.Also the operating system sees only a single process. A multithreaded system cannot make use of the different processors in a multiprocessor system simultaneously. The operating system will not schedule the different threads of the process on separate processors. There is no performance benefit associated with executing multiple user-level threads on a multiprocessor system
Another ans> No. The kernel is not aware of the user-level threads that are created. Therefore, it is not able to run the user-level threads on different processors.

4.9

When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multi-threaded solution would perform better even on a single-processor system.

4.10

Heap memory & Global variable

4.13

In a parallel system, two tasks must be performed simultaneously. Thus, it is possible to have concurrency without parallelism. This is shown in single core systems were The CPU scheduler rapidly switches between processes execution which allows all tasks to make progress but are not working in parallel

4.15

1. task
2.task (data)
3.data (task)
4.data
5.task

4.17

a. 6 (including parent)
b. 2

4.19

Child: 5 Parent: 0

4.20

Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

i. Means that some executing threads may have to wait to get a kernel process when needed. Would slow the system.
b. The number of kernel threads allocated to the program is equal to the number of processing cores.
i. Means that all executing processes will have a kernel thread they can access when needed. But when a kernel thread is blocked then the next thread that get schedules it will not have a kernel thread available and it could execute but will have to wait. So this too will slow performance
c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.
i. Means that all executing threads and others will have assigned kernel threads so it increases the likelihood that an executing thread will have a kernel thread to use when needed. This will improve performance.

And some: https://www.cram.com/flashcards/os340-chapter-4-5189218

chap-5

5.11
For CPU bound non voluntary context switches we require.

For I /O bound we require voluntary context switch

Because for voluntary context switch is used when process reqire resourse which more in case of i/o bound. Where as non voluntary context switch is used when time slice expire or higher priority process comes.

5.12
A. CPU utilization and response time: CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could, however, result in increasing the response time for processes.
B. Average turnaround time and maximum waiting time: Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could, however, starve long-running tasks and thereby increase their waiting time.

C. I/O device utilization and CPU utilization: CPU utilization is maximized by running long-running CPU-bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O-bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

5.13

By assigning more lottery tickets to higher-priority processes

5.14

The primary advantage of each processing core having its own run queue is that there is no contention over a single run queue when the scheduler is running concurrently on 2 or more processors. When a scheduling decision must be made for a processing core, the scheduler only need to look no further than its private run queue. A disadvantage of a single run queue is that it must be protected with locks to prevent a race condition and a processing core may be available to run a thread, yet it must first acquire the lock to retrieve the thread from the single queue. However, load balancing would likely not be an issue with a single run queue, whereas when each processing core has its own run queue, there must be some sort of load balancing between the different run queues.

5.15

When $\alpha = 0$ and $\tau_0 = 100$ milliseconds, the formula always makes a prediction of 100 milliseconds for the next CPU burst. When $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memoryless, and simply predicts the length of the previous burst forthe next quantum of CPU execution.

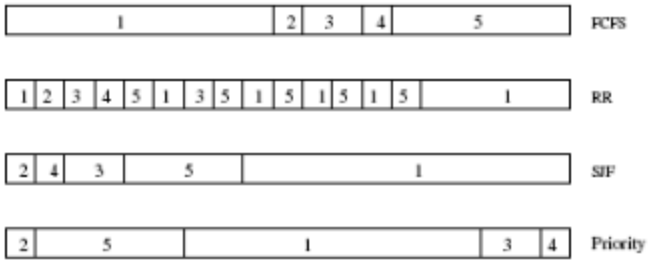
5.16

This scheduler would favor CPU-bound processes as they are rewarded with a longer time quantum as well as priority boost whenever they consume an entire time quantum. This scheduler does not penalize I/O-bound processes as they are likely to block for I/O before consuming their entire time quantum, but their priority remains the same

5.17

Answer:

a. The four Gantt charts are



b. Turnaround time

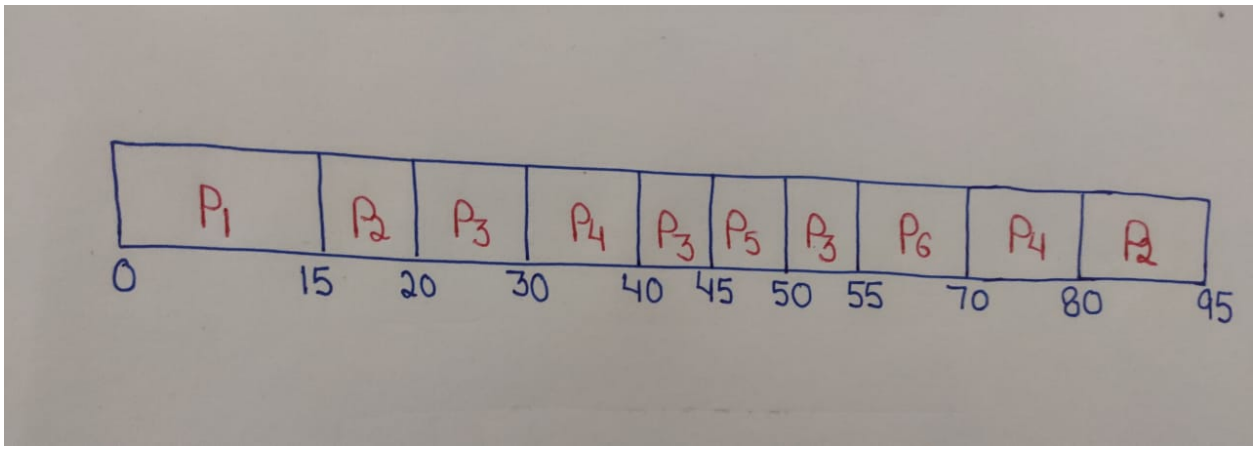
	FCFS	RR	SJF	Priority
P_1	10	19	19	16
P_2	11	2	1	1
P_3	13	7	4	18
P_4	14	4	2	19
P_5	19	14	9	6

c. Waiting time (turnaround time minus burst time)

	FCFS	RR	SJF	Priority
P_1	0	9	9	6
P_2	10	1	0	0
P_3	11	5	2	16
P_4	13	3	1	18
P_5	14	9	4	1

d. Shortest Job First

5.18



Process	Burst Time(BT)	Arrival Time(AT)	Completion Time (CT)	Turn Around time = CT - AT	Waiting time= TAT - BT
P1	15	0	15	15	0
P2	20	0	95	95	75
P3	20	20	55	35	15
P4	20	25	80	55	35
P5	5	45	50	5	0
P6	15	55	70	15	0

Average Turn Around Time = (15 + 95 +35 +55 + 5 +15)/6 = 220 / 6 = 36.67
Average Waiting Time = (75 + 15 + 35) / 3 = 125/3 = 41.67

5.19
Nice values < 0 are assigned a higher relative priority and such systems may not allow non-root processes to assign themselves higher priorities.

5.20
Shortest job first and priority-based scheduling algorithms could result in starvation.

5.21
a.In effect, that process will have increased its priority since by getting time more often it is receiving preferential treatment.
b.The advantage is that more important jobs could be given more time, in other words, higher priority in treatment. The consequence, of course, is that shorter jobs will suffer.
c.Allot a longer amount of time to processes deserving higher priority. In other words, have two or more quantums possible in the Round-Robin scheme.

5.22
(a) Time quantum is 1 ms.

Whether a CPU bound or I/O bound process, it switches every one millisecond and when doing so, it incurs a 0.1 ms overhead. Thus, for every 1.1 ms, the CPU is actually utilized only 1 ms.

So CPU utilization is (1 / 1.1)∗100 = 91%

(b) Time quantum is 10 ms.

Here, there is a difference between CPU bound and I/O bound processes. A CPU bound process can use the full 10 ms time slot, whereas an I/O bound process can have it only for 1 ms because another I/O bound process in the queue will snatch the time from it.

So a CPU bound process takes 10 ms, 10 I/O bound processes would take 10*1=10ms.

So, the CPU would be utilized for a total of 20 ms out of 21.1 ms. (Total time is 10*1.1 + 10.1=21.1ms).

Thus the CPU utilization is (20 / 21.1) ∗100 = 95%

5.23
Increasing the priority associated with the process

5.24
a.FCFS
b.LIFO

5.25
a) FCFS:here the cpu scheduling was based on first come, first served.that is here the CPU will be allocated as basis of the request available on the fifo queue. And if the cpu is being free the process is use the cpu until it complete its execution. And also as by the above mentioned rule if a larger program started to use the CPU, then all the other processes including the short process should be wait until it completes its execution. So it's clear that FCFS is discriminate the short process as by it have to wait if a large process uses the CPU.

b) RR:ROUND ROBIN mechanism, as by this mechanism there is no rule that a process can use the cpu until it completes, but here each process can equally use the cpu for a period of time called quantum. So if a process completes its execution in a quantum of time, then it can leave, else it have to be wait until all the other available processes completes its running for a quantum of time. So if the period of time is larger then it will behave as same as the FCFS, and else if the quantum of time is lesser then it treats all the process in a same basis.

c) multi level feedback queue :this scheduling method will allocate the CPU to a process as basis of their arrival and burst time. That is if the process need the CPU for its execution for a larger amount of time, then its priority will be settled to low, and otherwise to high. So this process

have to wait until the higher priority process uses the CPU, also if a process wait for the cpu for a larger amount of time, then its priority will automatically switched to high, so as this basis this mechanism also work similar to RR. and there is no such negative discrimination towards short jobs

5.26

SMP is basically Symmetric multiprocessing in which multiple processes are running at a same time which often causes inefficiency the reason why shared ready queue will suffer in performance because the SMP causes data loss and shared queue shares its resources with SMP causing inefficiency and will cause more time and resources so that's why we don't use shared ready queue for SMP environment because it creates lot of performance related issue and risk of data loss is more than in normal occasion

5.27

Solution is very simple.

1. Give higher priority to queue containing high priority thread and hence process the thread in this queue first.
2. Then once an element from high priority queue is dequeued, if balance between number of threads in these two queues are disturbed, then dequeue one thread from queue containing low priority thread and enqueue this into queue having high priority thread to balance the number of threads in these 2 queues. Since priority queue automatically adjust the thread within the queue based on its priority, hence removal of thread from one priority queue to another is not a problem.
3. If a new thread arrive with certain value of priority, then new thread could be added into queue containing less number of thread to maintain balance.

Thus in this way, load-balancing criteria of maintaining almost equal number of thread will be maintained and also giving more priority to high priority thread will also be maintained.

An additional thing to add in above statement is that, priority based scheduler, always check the priority of front element in both the queue. Hence scheduler will dequeue the thread from the queue containing highest priority thread. And then it is task of load-balancing algorithm to maintain balance of number of thread using method we discussed in 3 points above.

5.28

(i). If the new process is placed in the same queue as its parent , then it is just because of some similar strategy which is being followed by all the process in that queue . Although there will be

queue for the new process , but there will be a surity that all the processes will be processed smoothly without getting hanged at any point until the system throws some error . So having similar type os strategy , it is beneficial for the new process to be with its parent process .

(ii). If the new process is being scheduled in a different queue , it will be just for the shake of being reducing the time that will be taken by the process . There is some small chances that the process will be processed slowly i.e execution time can be slow but the compilation time or the taken taken by a process in the queue of its parent process will be much more as compared to that of the process placed in different queue . So for reducing time , the system might schedule the new process in a difefrent queue .

5.29

NUMA -aware scheduling algorithm should reschedule The thread on the same cpu on which it previously ran because it supports the spot of processes to memory and by dispatching the process on the very same node and then it can be done to process the dispatching processes on to the same processor. And also the thread in I/O spent a significant part of the time of excecution and also waits for the data from the storage to citation. And also the thread are those that the os in not aware of. But when the the I/o process finally completes now it will not excecute for a very long time before they start the next i/o operation.

But when the cpu thread are blocked then they cannot be run for a long time and then that is why the NUMA scheduling should reschedule the thread on the same cpu.

5.36

Dispatch latency is the cost associated with stopping one process and starting another. Both interrupt and dispatch latency needs to be minimized in order to ensure that real-time tasks receive immediate attention. Furthermore, sometimes interrupts are disabled when kernel data structures are being modified, so the interrupt does not get serviced immediately. For hard real-time systems,the time-period for which interrupts are disabled must be bounded inorder to guarantee the desired quality of service.

Chap-6

6.8

If there are two threads T1 and T2, which simultaneously try to change the value of highestBid, then the race condition can occur.. Consider an example with values, where currentHighestBid is 100. T1 tries to place amount 150 as bid, while T2 tries to place 200 as the bid..

Now lets assume execution starts with thread T1 and T1 only executes the statement if(amount > highestBid)
This statement is true, as amount = 150 for T1, hence T1 assumes its condition is met and will set the highestBid. however before it could set the variable, the thread is preempted by OS and execution is handed over to Thread T2.. So right now current highestBid is still at 100.

now T2 starts execution and T2 checkes for if(amount > highestBid)
This statement is true again for T2, as amount = 200 for T2, hence T2 sets the highest bid to 200..

now execution is back to T1 and T1 tries to finish the remaining part of the code. as it has done the comparison before itself, hence it now just sets its amount(150) to the highest bid.. So highestBid becomes 150 now..

As we have seen, two thread place 150 and 200 as bids, and bid should have becomes 200, while here because of the race condition, the result comes out to be 150// which is wrong.

To prevent the race condition, The method should be marked synchronized and should be executed atomically.

6.9

The given algorithm has race conditions since values[k] variable can be accessed by all processors where values[k] is changing frequently at a time which will lead to race conditions and deadlocks.

To prevent this dead locking or racecondtions,you need to use
atomic variables or mutex locks so that values[k] can be accessed

only once at a time.

```
for j = 1 to log_2(N) {

for k = 1 to N {

if ((k + 1) % pow(2,j) == 0) {

values[k] += values[k - pow(2,(j-1))]

}

}
```

6.10

The given usage of compare_and_swap() instruction (CAS) can't be totally liberated from race conditions unless the atomicity of the statement is guaranteed.

Node *top; /head of stack

The execution has no mention whether the access to the shared Node is strictly atomic in nature. If it is so, no race condition happens while performing the PUSH or POP operations. Otherwise, multiple processes may unitentionally access the stack and perform the PUSH or POP operations,hence prompts race condition.

6.11

The idiom compare and compare-and-swap works appropriately for implementing spinlocks. But it is just like compare- and-swap with an extra initial condition to be checked. Other than this there is no difference in the functionality to get the mutual inclusion between the processes on a resource.

6.12

```
if ( getValue (&sem) >0 )
    wait(&Sem) ;
```

⇒ Let initially value of sem=1.

Let P_1 be the first process to execute. Thus P_1 will first process to perform getValue (&sem).

This will be true, thus it will perform wait(&sem).

But before that P_2 will preempt P_1.

P_2 will now execute getValue (&Sem) and this will be TRUE and therefore it will perform wait(&sem).

Thus P_2 will now decrement value of sem = 0.

Now P_1 will starts its execution from the point it left. Now it will perform wait (&Sem). Since sem=0, P_1 performing wait on sem will get blocked.

The idea here was to prevent process from blocking, but due to interleaving between process, there is a chance of process getting blocked.

6.13

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.

6.15
implementing synchronization primitives by disabling interrupts is not so appropriate in a single-processor system,If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes to execute

6.16
```
// initialization
mutex->available = 0;


// acquire using compare and swap()
void acquire(lock *mutex) {
while (compare_and_swap(&mutex->available, 0, 1) != 0)
;     //implement more if you want
return;
}


// acquire using test and set()
void acquire(lock *mutex) {
while (test_and_set(&mutex->available) != 0)
;   //implement more if you want
return;
}

//release the lock
```

```
void release(lock *mutex) {
mutex->available = 0;
return;
}
```

6.17

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

6.18

Onno akta question

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

Ans.

```
int guard = 0;
int semaphore value = 0;
wait()
{
while (TestAndSet(&guard) == 1);
if (semaphore value == 0) {
atomically add process to a queue of processes
waiting for the semaphore and set guard to 0;
} else {
semaphore value--;
guard = 0;
}
}
signal()
{
while (TestAndSet(&guard) == 1);
if (semaphore value == 0 &&
there is a process on the wait queue)
wake up the first process in the queue
of waiting processes
else
semaphore value++;
guard = 0;
```

}

6.19
The lock is to be held for a short duration: It makes more sense to use a spinlock as it may in fact be faster than using a mutex lock which requires suspending, awakening and the waiting process.

The lock is to be held for a long duration: A mutex lock is required as this allows the other processing core to schedule another process while the locked process waits.

A thread may be put to sleep while holding the lock: A mutex lock is required as you do not want the waiting process to be spinning while waiting for the other process to wake up.

6.20

6.21
First Method

int hits;
mutex lock hit lock;
hit lock.acquire();
hit lock.release();

System call is required for lock and it will put a process to sleep. So, it requires context switching. If the lock is unavailable.

The process awakening will also require another subsequent context switch.

Second Method

atomic t hits;
atomic inc(&hits);

Atomic update of hits variables are provided by atomic integer.
It also ensures no race condition on hits.
No kernel intervention is need in this case.

Therefore the second method is more efficient than the first one.

6.22

a) Race condition would come when any process tries to run statements

```
++number of processes;  --number of processes;
```

b) To prevent race condition

acquire() lock will be used when process tries to perform

```
++number of processes;
```

similarly release() locak will be used when process tries to perform

```
--number of processes;
```

c) Yes, atomic integer variable can be used to prevent race condition where at a time only one process can increment or decrement the variable number of process.

[note: acquire() lock and release() lock use before and after of the operations respectively]

6.23

Semaphores are one type of Interprocess Communication. That is, Semaphores provide a mechanism for one application, process, or thread to communicate with other different applications,processes, or threads. Semaphores can be used to monitor and control the number of concurrent open socket connections by allowing each process which creates an open socket connection to communicate the open connection's existence and properties to other processes.

If the server knows about each such open connection via this semaphore communication, the server can then instruct each of the individual processes to either quit making new connections or to continue making new connections until the limit is reached.

Chap-8

8.13

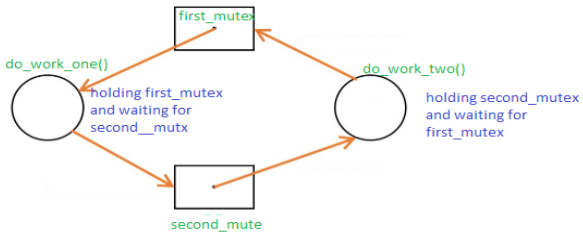
```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
```

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
```

The second thread executed:

Hence now, both are holding one and one resource as first_mutex and second_mutex. Now, its a deadlock condition because they will wait on each other to release the resources.

Let's draw the diagram of the Resource allocation graph:



8.15

YES.

(1)Mutual exclusion is maintained, as they cannot be shared if there is a writer.

(2)Hold-and-wait is possible, as a thread can hold one reader—writer lock while waiting to acquire another.

(3) You cannot take a lock away, so no preemption is upheld.

(4) A circular wait among all threads is possible

8.16

If thread_one is scheduled before thread_two and thread_one is able to acquire both mutex locks before thread_two is scheduled, deadlock will not occur. Deadlock can only occur if either thread_one or thread_two is able to acquire only one lock before the other thread acquires the second lock

8.18

Soln link: https://9lib.co/document/q05p12kx-operating-system-homework.html

8.19

A deadlock-avoidance scheme tends to increase the runtime overheads due to the cost of keep track of the current resource allocation. However, a deadlock-avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. In that sense, a deadlock avoidance scheme could increase system throughput

8.20

a. Increase Available (new resources added)—This could safely be changed without any problems.

b. Decrease Available (resource permanently removed from the system)—This could have an effect on the system and introduce the possibility of deadlock as the safety of the system assumed there were a certain number of available resources.

c. Increase Max for one process (the process needs more resources than allowed, it may want more)—This could have an effect on the system and introduce the possibility of deadlock.

d. DecreaseMax for one process (the process decides it does not need that many resources)—This could safely be changed without any problems.

e. Increase the number of processes—This could be allowed assuming that resources were allocated to the new process(es) such that the system does not enter an unsafe state.

f. Decrease the number of processes—This could safely be changed without any problems.

8.21

Using the terminology of Section Section 7.6.2, we have: a. $\sum i < m + n$ b. $Max_i \geq 1$ for all i Proof: $Need_i = Max_i - Allocation_i$ If there exists a deadlock state then: c. $\sum i = m$ Use a. to get: $\Sigma Need_i + \Sigma Allocation_i = \Sigma Max_i < m + n$ Use c. to get: $\Sigma Need_i + m < m + n$ Rewrite to get: $\sum i < n$ This implies that there exists a process Pi such that $Need_i = 0$. Since $Max_i \geq 1$ it follows that Pi has at least one resource that it can release. Hence the system cannot be in a deadlock state.

8.23

Suppose N = Sum of all Need(i), A = Sum of all Allocation(i), M = Sum of all Max(i). Use contradiction to prove.
Assume this system is not deadlock free. If there exists a deadlock state, then A = m because there's only one kind of resource and resources can be requested and released only one at a time. From condition b, N + A = M < m + n. So we get N + m < m + n. So we get N < n. It shows that at least one process i that Need(i) = 0. From condition a, Pi can release at least 1 resource. So there are n-1 processes sharing m resources now, condition a and b still hold. Go on the argument, no process will wait permenently, so there's no deadlock.

8.24

The following rule prevents deadlock: when a philosopher makes a request for the first chopstick, do not grant the request if there is no other philosopher with two chopsticks and if there is only one chopstick remaining.

8.25

8.28
SOLN: https://9lib.co/document/q05p12kx-operating-system-homework.html

AND SOME ANS CHAPTER WISE:
https://studylib.net/doc/25466468/abraham-silberschatz-operating-system-concepts-ch-2-8-sol…

Chap-9

9.11
Internal fragmentation
Definition
Internal fragmentation occurs when there is unused space within one of the partitions of memory allocated to a process

Schemes that suffer from internal fragmentation
Fixed-size memory allocation schemes

Solving internal fragmentation
Internal fragmentation can be reduced by using multiple variable-sized partitions

When unused space can be used
The space that is allocated to a process and is unused cannot be used by the system until the process releases it

External Fragmentation
Definition
External fragmentation occurs when there is enough free space to satisfy a request for memory but there is not enough contiguous available memory to be allocated to a process

Schemes that suffer from external fragmentation
Multiple partition allocation schemes

Solving external fragmentation
External fragmentation can be solved using compaction

When unused space can be used
The space that is allocated to a process and is unused can be used by the system after compaction is performed

9.12
Address binding of instructions and data to memory addresses can happen at three stages: Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes. Load time: Must generate relocatable code if memory location is not known at compile time.

The linkage editor has to exchange unresolved symbolic addresses with the actual addresses associated with the variables in the binary code.

To do this, the modules should save instructions that refer to unresolved symbols.

During linking, each module is assigned a sequence of addresses in the overall program binary and when this has been performed, unresolved references to symbols exported by this binary could be patched in other modules since every other module would contain the list of instructions that need to be connected.

9.13
The three ways to assign a partition or hole to the process is given below:

First Fit
Best Fit
Worst Fit
The first fit and the worst fit can't satisfy the given memory request.

First Fit:

The first partition that is large enough is assigned to the process.

The memory allocation for the above approach is given below:

Partiopartitions	100 MB	170 MB	40 MB	205 MB	300 MB	185 MB
Process size	15 MB	75 MB		200 MB	185 MB	175 MB
	80 MB X					

There is no space for the process to have a size of 80 MB.

Best Fit:

The partition is just greater than or equal to the process.

The memory allocation for the above approach is given below:

Partiopartitions	100 MB	170 MB	40 MB	205 MB	300 MB	185 MB
Process size	75 MB	80 MB	15 MB	200 MB	175 MB	185 MB

205 MB Partial Partitions Process size 100 MB 75 MB 170 MB 80 MB 40 MB 15 MB 300 MB 175 MB 185 MB 185 MB 200 MB

All processes will allocate space in this approach.

Worst Fit:

The partition that is the largest is assigned to the process.

The memory allocation for the above approach is given below:

Partiopartitions	100 MB	170 MB	40 MB	205 MB	300 MB	185 MB
Process size	80 MB	75 MB		15 MB	200 MB	185 MB
	175 MB X					

40 MB Partial Partitions 100 MB Process size 80 MB 175 MB x 170 MB 75 MB 205 MB 15 MB 300 MB 200 MB 185 MB 185 MB

There is no space for the process to have a size of 175 MB.

None of the algorithms is best and it entirely depends upon the situation.

The experiments have shown that the best fit is best for MFT(Multiprogramming Fixed number of Tasks).

9.14

Contiguous-memory allocation:

- This schema requires relocation of the entire program,
- Explanation: Since there is no enough required space for the program to grow beyond it's allocated memory space, it may require relocation of entire program.

Pure segmentation:

- This schema requires relocation of the segment that needs to be extended,
- Explanation: Since there is no enough required space for the segment to grow beyond it's allocated memory space, it may require relocation of segment.

Pure paging:

- This schema requires incremental allocation of new pages
- Explanation: It is possible in this scheme without requiring relocation of the program's address space.

9.25

a) 2 memory accesses: page lookup followed by actual access => 2*50ns = 100ns

b) 75% * (TLB hit-time + memory access time) + 25% * (TLB hit-time + 2*memory access time)= 75% * (2+50 ns) + 25% * (2+ 2*50 ns) = 64.5 ns

The percentage of times that a particular page number is found in the TLB is called the **hit ratio.** An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the TLB.

If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the **effective memory-access time,** we must weigh each case by its probability:

EAT(effective access time)= P x hit memory time + (1-P) x miss memory time.

effective access time = 0.80 x 120 + 0.20 x 220
= 140 nanoseconds.

In this example, we suffer a 40-percent slowdown in memory access time (from 100 to 140 nanoseconds).

Chap-10

10.15

a) TLB miss with no page fault
It is the case when word is not present in CPUcache but present in main memory(RAM).

b) TLB miss and page fault
It is the case when word is not present in CPUcache and as well as not present in main memory(RAM). It is inlocal hard disk(Swapped out).

c) TLB hit and no page fault

It is the case when word is present in CPU cache.

d) TLB hit and page fault
It is not possible.
Reason: if there is TLB hit, then this word will be found in CPUcache and no need to refer the main memory for Page.

10.16
a. If a page fault occurs, then the thread will changes it state from running to blocked. Because page fault means the required page is not in the main memory, so the scheduler has to load that page from secondary memory to main memory. Therefore here the thread is waiting for I/O operation to be performed. Hence it will change its state from running to blocked.

b. TLB means translation look aside buffer. It contains the details of the page table entries which we used recently. TLB is used to reduce to time for accessing pages in main memory. Even if we get a TLB miss it doesnt mean that the page is not there in main memory, it only means that the page is not used recently. So here we dont need to perform any I/O operation. Therefore the thread wont change its state.

c. The thread wont change its state even if the address reference is resolved in the page table, because here there is no such need to perform an I/O operation. So it continues its state of running.

10.17
a. When there are no pages which could be used by process is present in memory then it will generate page fault. Initially when process starts execution, we can characterize the page-fault rate equal to number of page fault occured divided by number of instruction executed. Hence more the number of instructions present per page, lesser will be page fault rate.

b. Once the entire working set of pages is loaded, then page fault can occur only if number of pages require to execute the process cannot be accomodated into physical memory at the same time. Hence the page-fault rate will be equal to number of page fault occured divided by number of pages accessed by the process. Hence page-fault rate will be less if more number of pages are accessed with less number of page fault.

c. One of the option is to use virtual memory in which logical memory space is larger than primary memory and hence some part of program not currently being used by process resides in the secondary memory and hence based on demand, the pages from secondary memory can be fetched to primary memory.

Another thing is to use global page replacement policy where process can replace the frames used by other process and hence this process can use more physical memory space on demand if other processes in the system does not require much space.

10.18

Page size = 256B -> Page offset= log 256 = 8 bit.

Virtual address (12 bit)

Virtual Page Number (4 bit)	Page Number (8 bit)

Virtual address and physical address will have same page offset. Only their page number or frame number will change.

1)

A) 0x2A1 -> 0010 1010 0001

Here page number => 0010 => 2

Page offset = 0xA1

Look into page table for entry of page # 2. The corresponding frame number is 0xA.

The corresponding physical address is :

Physical address

Frame number 0xA	Frame offset 0xA1

Physical address = 0xA1A

** **OxAA1**

B) 0x4E6

Here page number = 0x4

Page offset = 0xE6

The page table entry for page number 4 has corresponding frame number as invalid.

Therefore this is a Page Fault.

Bring the corresponding required page. The frame number then would be 0x9.

Therefore physical address : 0x9E6

C) 0x94A

Page number= 0x9

Page offset = 0x4A

The corresponding frame number from page table is 0x1

Thus physical address is 0x14A

D) 0x316

Page number = 0x3

Page offset = 0x16

This is a page fault since the entry in page table is invalid.

Bring the required page into main memory and update page table with frame number with 0xF.

The physical address is 0xF16

10.19

Copy on Write allows processes to share pages rather

than each having a separate copy of the pages. However,

when one process tried to write to a shared page, then

a trap is generated and the OS makes a separate copy of the

page for each process. This is commonly used in a fork()

operation where the child is supposed to have a complete

copy of the parent address space. Rather than create

a separate copy, the OS allows the parent and child

to share the parent's pages. However, since each is supposed

to have its own private copy of the pages, the pages

are copied when one of them attemps a write.

The hardware support required to implement is simply the following:

on each memory access, the page table needs to be consulted to

checkwhether the page iswrite-protected. If it is indeedwrite-protected,

a trap would occur and the operating system could resolve the issue.

10.20

Assuming the virtual memory space to be 232 as 212 does not make any sense for a computer with 222 bytes of physical memory.

Firt of all, Let us write the virtual address i.e. 11123456 in binary form. It is represented as:

0001 0001 0001 0010 0011 0100 0101 0110.

Now since the page size is 4096 = 212 , the page table size will be 2(32-12) = 220 .

Therefore the low order 12 bits from the virtual address given to us i.e 0100 0101 0110 are used as the displacement into the page, while the remaining 20 bits of the virtual address from the left i.e. 0001 0001 0001 0010 0011 are used as the displacement in the page table.

The offset bits are then concatenated to the resulting physical page number i.e. from the page table to form the actual final address.

The logical address calculation parts are software operations, while accessing the actual physical address from the memory are the hardware operations.

10.21

Solution:

10.5

- Time to service a page fault =8 milliseconds.
- Time to modify the replaced page = 20 milliseconds.
- Memory access time = 100 nano seconds.
- Page replacement modification time percentage is 70%.
- Effective Memory Access Time = $(1 - p) \times$ memory access time + $p \times$ page fault service time
- The p is the page fault rate
- $(1 - p) \times 100 \text{ ns} + p \times \{(.70 \times 20 + .30 \times 8) \times 106 \text{ ns}) \leq 200 \text{ ns}$
- $= 100 - 100p + 16400000p \leq 200$
- $= 16399900p \leq 100$
- $p \leq 1/163999$
- $p = 0.000006$

**10^6 ns

10.22

Size of page = size of frame = 4096 bytes

Number of bits in offset = log2 (size of page) = log2 4096 = 12 bits

Hence out of 16 bit virtual address, first 4 bits will be page number while last 12 bits will be page offset.

Hence physical address of a virtual address is determined by checking for the page frame number corresponding to page number and then appending the 12 bits offset.

a. 0x621C :- Here the page number is leftmost first digit of virtual address. So page number is 6 and offset is 21C. Since page frame number is 8. Hence the physical address is 0x821C and Reference bit of page 6 will be set to 1.

b. 0xF0A3 :- Page number is F and offset is 0A3. Since page frame number is 2. Hence the physical address is 0x20A3 and Reference bit of page F(15 in decimal) will be set to 1.

c. 0xBC1A :- Page number is B and offset is C1A. Since page frame number is 4. Hence the physical address is 0x4C1A and Reference bit of page B will be set to 1.

d. 0x5BAA :- Page number is 5 and offset is BAA. Since page frame number is 13. Hence the physical address is 0xDBAA and Reference bit of page 5 will be set to 1.

e. 0x0BA1 :- Page number is 0 and offset is BA1. Since page frame number is 9. Hence the physical address is 0x9BA1 and Reference bit of page 0 will be set to 1.

b. We can see that logical page like 1, 9 and 14 are not present in physical memory and hence any reference to them will result in page fault. So take an address like 0x9A2F will result in page fault.

c. LRU page-replacement algorithm will replace the page frame which has been reference list recently. Here the physical page frame 3, 11 and 14 are not present in page table and hence they are least recently used. Hence LRU page replacement will select the page from page set {3, 11, 14}.

10.23

Summary - FIFO algorithm

- Total frames: 3
- Algorithm: FIFO
- Reference string length: 20 references
- String: 4 2 1 7 9 8 3 5 2 6 8 1 0 7 2 4 1 3 5 8

Solution visualization

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ref		**4**	**2**	**1**	**7**	**9**	**8**	**3**	**5**	**2**	**6**	**8**	**1**	**0**	**7**	**2**	**4**	**1**	**3**	**5**	**8**
f		4	2	1	7	9	8	3	5	2	6	8	1	0	7	2	4	1	3	5	8
f			4	2	1	7	9	8	3	5	2	6	8	1	0	7	2	4	1	3	5
f				4	2	1	7	9	8	3	5	2	6	8	1	0	7	2	4	1	3
hit		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
v					4	2	1	7	9	8	3	5	2	6	8	1		7	2	4	1

- Total references: 20
- Total distinct references: 10
- Hits: 0
- Faults: 20
- **Hit rate:** 0/20 = **0**%
- **Fault rate:** 20/20 = **100**%

Summary - LRU algorithm

- Total frames: 3
- Algorithm: LRU
- Reference string length: 20 references
- String: 4 2 1 7 9 8 3 5 2 6 8 1 0 7 2 4 1 3 5 8

Solution visualization

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ref		**4**	**2**	**1**	**7**	**9**	**8**	**3**	**5**	**2**	**6**	**8**	**1**	**0**	**7**	**2**	**4**	**1**	**3**	**5**	**8**
f		4	2	1	7	9	8	3	5	2	6	8	1	0	7	2	4	1	3	5	8
f			4	2	1	7	9	8	3	5	2	6	8	1	0	7	2	4	1	3	5
f				4	2	1	7	9	8	3	5	2	6	8	1	0	7	2	4	1	3
hit		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
v					4	2	1	7	9	8	3	5	2	6	8	1		7	2	4	1

- Total references: 20
- Total distinct references: 10
- Hits: 0
- Faults: 20
- **Hit rate:** 0/20 = **0**%
- **Fault rate:** 20/20 = **100**%

Summary - OPT algorithm

- Total frames: 3
- Algorithm: OPT
- Reference string length: 20 references
- String: 4 2 1 7 9 8 3 5 2 6 8 1 0 7 2 4 1 3 5 8

Solution visualization

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ref		**4**	**2**	**1**	**7**	**9**	**8**	**3**	**5**	**2**	**6**	**8**	**1**	**0**	**7**	**2**	**4**	**1**	**3**	**5**	**8**
f		4	2	1	7	9	8	3	5	5	6	6	1	0	7	7	4	4	3	5	8
f			4	2	1	1	1	8	8	8	8	8	8	1	1	1	1	1	1	1	1
f				4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
hit		✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗
v					4	7	9	1	3		5		6	8			7		4	3	5

- Total references: 20
- Total distinct references: 10
- Hits: 4
- Faults: 16
- **Hit rate:** 4/20 = **20**%
- **Fault rate:** 16/20 = **80**%

10.24

10.36
EMAT
= 80% of Reference from associative memory + 18% form page table + 2% from page fault
= (0.8 * 1 us) + (0.18 * 2 us) + (0.02 * (20000us + 2us))
= 0.8 us + 0.36 us + 400.04 us
= 401.2 us

10.37
Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

Jurassic park:

Passengers = m processes

Cars = n processes

This program is similar to the barber shop problem. P(Passenger_released) program same as semwait andV(Passenger_released) same as sensignal methods.

In process passenger (i = 1 to num_passengers)

P(passenger released);

In process Car (j = =1 to num_cars)

V (Passenger_released);

When processor car V(Passenger_released) is match up with aprocess passenger P(Passenger_released) for a passenger of another car.

A car can release a passenger then that is not the passenger– riding inside its car.

So for this fixing requires an array of semaphores. (One per car or one per passenger).