

Solution Operating System Concepts By Galvin, Silberschatz

Solved By Abhishek Pharkya

Part 1: Theory

What is the primary difference between a kernel-level context switch between processes (address spaces) and a user-level context switch?

The primary difference is that kernel-level context switches involve execution of OS code. As such it requires crossing the boundary between user- and kernel-land two times. When the kernel is switching between two different address spaces it must store the registers as well as the address space. Saving the address space involves saving pointers to the page tables, segment tables, and whatever other data structures are used by the CPU to describe an address space. When switching between two user-level threads only the user-visible registers need to be saved and the kernel need not be entered. The overhead observed on a kernel-level context switch is much higher than that of a user-level context switch.

Does spawning two user-level threads in the same address space guarantee that the threads will run in parallel on a 2-CPU multiprocessor? If not, why?

No, the two user-level threads may run on top of the same kernel thread. There are, in fact, many reasons why two user-level threads may not run in parallel on a 2-CPU MP. First is that there may be many other processes running on the MP, so there is no other CPU available to execute the threads in parallel. Second is that both threads may be executed on the same CPU because the OS does not provide an efficient load balancer to move either thread to a vacant CPU. Third is that the programmer may limit the CPUs on which each thread may execute.

Name three ways to switch between user mode and kernel mode in a general-purpose operating system.

The three ways to switch from between user-mode and kernel-mode in a general-purpose operating system are in response to a system call, an interrupt, or a signal. A system call occurs when a user program in user-space explicitly calls a kernel-defined "function" so the CPU must switch into kernel-mode. An interrupt occurs when an I/O device on a machine raises an interrupt to notify the CPU of an event. In this case kernel-mode is necessary to allow the OS to handle the interrupt. Finally, a signal occurs when one process wants to notify another process that some event has happened, such as that a segmentation fault has occurred or to kill a child process. When this happens the OS executes the default signal handler for this type of signal.

Consider a unprocessed kernel that user programs trap into using system calls. The kernel receives and handles interrupt requests from I/O devices. Would there be any need for critical sections within the kernel?

Yes. Assume a user program enters the kernel through a trap. While running the operating system code, the machine receives an interrupt. Now, the interrupt handler may modify global data structures that the kernel code was trying to modify. Therefore, while

there is only one thread that runs inside the kernel at any given time, the kernel may not be re-entrant if access to global data structures is not protected through the use of appropriate mutexes.

Name the pros and cons of busy waiting. Can busy waiting be avoided altogether in synchronization primitive?

One of pros of busy waiting is that it is efficient in cases where the expected wait time is less than the overhead of context switching out of and back to the waiting thread. This happens when a critical section is protecting just few lines of code. It is also good in that a thread can simply stay on the CPU rather than having to give up the CPU before its quantum expires. The biggest con of busy waiting is that it burns CPU cycles without accomplishing anything. By definition a busy wait just spins on the CPU until the lock becomes available, and perhaps these cycles could be used for some other computation. It is important to note that busy waiting is not ever good on a uniprocessor. If there is only one CPU in the system, then there is no chance that the lock will be released while the thread is spinning. In the best case cycles are truly wasted, in the worst the system deadlocks. However, busy waiting can be highly effective on MPs. This leads to a second con: busy waiting leads to an important difference between UP and MP code.

Busy waiting can be avoided in synchronization primitive if the primitive always performs a yield whenever an unavailable lock is needed.

Can a mutual exclusion algorithm be based on assumptions on the relative speed of processes, i.e. that some processes may be "faster" than the others in executing the same section of code?

No, mutual exclusion algorithms cannot be based on assumptions about the relative speed of processes. There are MANY factors that determine the execution time of a given section of code, all of which would affect the relative speed of processes. A process that is 10x faster through a section of code one time, may be 10x slower the next time.

If processes do actually differ in their speed, does this pose any other problems on the system? Hint: Think of fairness.

It depends. Fairness in the eyes of the operating system is in terms of the resources which it provides to each process. As long as each process is given the same resources, then fairness is achieved regardless of the efficiency each thread has with respect to these resources. One problem that may come up if two threads execute at greatly different speeds is if they must periodically synchronize with each other. In this case the faster thread would have to wait for the slower thread to reach the checkpoint before it could continue.

Why does disabling interrupts affect the responsiveness of a system, i.e. the ability of the system to respond quickly to user requests?

Disabling interrupts reduces the responsiveness of a system because the system will be

unable to respond to any requests from external devices or programs for the duration that interrupts are disabled. If no interrupts are ever allowed then the only time the OS can check for the occurrence of events is during kernel-level context switches. This limits the granularity at which the OS can respond to user interaction. For example, if you press a key on your keyboard, the OS will not pass that message on to your text editor until the next context switch. This may cause applications to appear unresponsive. On the other hand if interrupts are rarely used, then the OS can temporarily suspend an executing process to handle an interrupt between nearly any two instructions, thereby causing good responsiveness.

Part 2: Problems

What is the output of the following programs? Please solve the problem without actually compiling and running the programs! We assume that you are familiar with the UNIX system calls `fork` and `wait`.

Program 1:

```
main ()
{
    val = 5;
    if(fork())
        wait(&val);
    val++;
    printf("%d\n", val);
    return val;
}
```

Program 2:

```
main()
{
    val = 5;
    if(fork())
        wait(&val);
    else
        exit(val);
    val++;
    printf("%d\n", val);
    return val; }
```

In the first program, the parent process creates a child and then waits for the child to exit (through the system call "wait"). The child executes and prints out the value of `val`, which is "6" after the `val++` statement. The child then returns the value of `val` to the parent, which receives it in the argument to "wait" (`&val`). The parent then prints out the value of `val`, which is now 7. Note that the parent and child have separate copies of the variable "val".

Using similar reasoning, you can see that the parent in program 2 waits for the child to return, and the child exits immediately. In this case only one value gets printed out which is the number 6 (from the parent process.)

Note: Problem from the fall'03 Midterm Write down the sequence of context switches that would occur in Nachos if the main thread were to execute the following code. Assume that the CPU scheduler runs threads in FIFO order with no time slicing and all threads have the same priority. The willjoin flag is used to signify that the thread will be joined by its parent. In your answer use the notation "childX->childY" to signify a context switch. For example "child1->child2" signifies a context switch from child1 to child2. Assume that the join causes the calling thread to release the CPU and sleep in some waiting queue until the child completes its execution.

```
void thread::SelfTest2() {  
    Thread *t1 = new Thread ("child 1", willjoin);  
    Thread *t2 = new Thread ("child 2", willjoin);  
  
    t1->fork((VoidFunctionPtr) &Thread::Yield, t1);  
    t2->fork((VoidFunctionPtr) &Thread::Yield, t2);  
    t2->join;  
    t1->join;  
}
```

This will cause the following sequence of context switches to occur:

main->t1->t2->t1->t2->main.

This sequence occurs because when the main thread fork () s two children they don't begin executing until main relinquishes the CPU which happens when main calls join (). Fork () has put t1 at the head of the run queue and t2 behind it, so when main calls join (), Nachos context switches to t1, causing the execution of t1. t1 executes Yield() and in so doing it puts itself at the tail of the run queue and relinquishes the CPU. When this happens Nachos looks for the next thread on the run queue, which is t2. So Nachos context switches to t2. As with t1, t2 yields the CPU and Nachos context switches back to t1. t1 finishes executing and exits. When this happens Nachos takes t2 off the run queue and context switches to it, allowing it to finish execution and exit. This whole time main has not been able to execute because it was not on the run queue because it was stuck in join () waiting for t2 to finish. Now that t2 has finished, Nachos context switches back to main which is able to pass the t1->join because t1 has already finished as well.

Prove that, in the bakery algorithm (Section 7.2.2), the following property holds: If P_i is in its critical section and P_k ($k \neq i$) has already chosen its number $[k] \neq 0$, then $(\text{number}[i], i) < (\text{number}[k], k)$.

Note: there are differences in the implementation of the bakery algorithm between the book and the lectures. This solution follows the book assumptions. It is not difficult to adapt the solution to cope with the assumptions made in the lecture notes. Suppose that P_i

is in its critical section, and $P_k(k \neq i)$ has already chosen its number[k], there are two cases:

1. P_k has already chosen its number when P_i does the last test before entering its critical section.

In this case, if $(\text{number}[i], i) < (\text{number}[k], k)$ does not hold, since they can not be equal, $(\text{number}[i], i) > (\text{number}[k], k)$. Suppose this is true, then P_i can not get into the critical section before P_k does, and P_i will continue looping at the last while statement until the condition does not hold, which is modified by P_k when it exits from its critical section. Note that if P_k gets a number again, it must be larger than that of P_i .

2. P_k has not chosen its number when P_i does the last test before entering its critical section.

In this case, since P_k has not chosen its number when P_i is in its critical section, P_k must choose its number later than P_i . According to the algorithm, P_k can only get a bigger number than P_i , so $(\text{number}[i], i) < (\text{number}[k], k)$ holds.

Too Much Milk: Two robotic roommates, A and B, buy milk using the following processes (note that the two roommates are not doing the exact same thing, and that their brains work using general-purpose microprocessors!):

```
NoteA=TRUE;
while(NoteB==TRUE) ;
if(NoteB==FALSE){
    if(NoMilk){
        BuyMilk();
    }
}
noteA=FALSE;
```

```
NoteB=TRUE;
if(NoteA==FALSE) {
    if(NoMilk) {
        BuyMilk();
    }
}
noteB=FALSE;
```

Is there any chance that the two roommates buy too much milk for the house ? If not, do you see any other problems with this solution?

This is a correct solution to the "Too Much Milk" problem. The robotic roommates will never buy too much milk. There are, however, other problems with this solution. First, it is asymmetric in that both roommates do not execute the same code, which is a problem. Second, it involves busy waiting by roommate A.

Part 1: Theory

Describe the difference between deadlock and starvation. The difference between deadlock and starvation is that with deadlock none of the threads in a set of threads are able to make progress because the events they are waiting for can only be triggered by other threads in the set that are also blocked. With starvation, some threads may make progress while others fail to ever make progress because for example, they are losing in all races for a semaphore. As another example, if a LIFO queue is used for locks, some threads may starve if they happen to end up at the bottom of the queue.

Why is repeated checking of condition variables needed upon wakeups? In other words, why are we using a while instead of an if when we invoke waits on a condition variable? When a thread wakes up from a CV it is because some condition has been made true, such as that a spot in a list has been made available. It is possible though that there are other threads in the system which could negate this condition before the thread in question has a chance to execute, for example if there are many producers in a producer/consumer problem. Just because a condition becomes true and a thread is woken up doesn't mean that that condition will be true when the thread is scheduled, so it is necessary to check to make sure that the condition is still true upon rescheduling. This could happen an unlimited number of times, so we have to check repeatedly.

Can semaphores be implemented with condition variables? Can condition variables be implemented with semaphores? Semaphores can be implemented with condition variables, provided that there is also a primitive to protect a critical section (lock/unlock) so that both the semaphore value and the condition are checked atomically. In Nachos for example, this is done with disabling interrupts.

CVs can be implemented with semaphores. You have solved this problem during your second programming assignment.

Define briefly the lost wakeup problem. The lost wakeup problem occurs when two threads are using CVs to synchronize their execution. If thread 1 reaches the case where the necessary condition will be false, such as when a consumer sees that the buffer is empty, it will go to sleep. It is possible that the OS will interrupt thread 1 just before it goes to sleep and schedule thread 2 which could make the condition for thread 1 true again, for example by adding something to the buffer. If this happens thread 2 will signal thread 1 to wake up, but since thread 1 is not asleep yet, the wakeup signal is lost. At some point thread 1 will continue executing and immediately go back to sleep. Eventually, thread 2 will find its condition to be false, for example if the buffer becomes full, and it will go to sleep. Now both threads are asleep and neither can be woken up.

Part 2: Problems

Problem comes from past midterm. You have been hired by your professor to be a grader for CSCI444/544. Below you will find a solution given by a student to a concurrency assignment. For the proposed solution, mark it either as i) correct, if it has no flaws, ii) incorrect, if it doesn't work, or if it works for some cases but not all cases. Assume condition variables with semantics as defined in the lectures.

If the solution is incorrect, explain everything wrong with the solution, and add a minimal amount of code to correct the problem. Note that you must not implement a completely different solution in this case -- use the code that we give you as a base.

Here is the synchronization problem: A particular river crossing is shared by Microsoft employees and Linux hackers. A boat is used to cross the river, but it only seats three people, and must always carry a full load. In order to guarantee the safety of the Linux hackers, you cannot put one Linux hacker and two Microsoft employees in the same boat (because the Microsoft guys will gang up and corrupt the pure mind of the Linux hacker), but all other combinations are legal. Two procedures are needed: `MSDudeArrives` and `LinuxDudeArrives`, called by a Microsoft employee or a Linux hacker when he/she arrives at the river bank. The procedures arrange the arriving MS and Linux dudes in safe boatloads; once the boat is full, one thread calls `RowBoat` and after the call to `RowBoat`, the three procedures then return. There should also be no undue waiting, which means that MS and Linux dudes should not wait if there are enough of them for a safe boatload.

Here's the proposed solution:

```
int numLinuxDudes = 0, numMSDudes = 0;

Lock *mutex;

Condition *linuxwait, *mswait;

void RowBoat{printf("Row, row, row your boat");}

void LinuxDudeArrives () {
    mutex->Acquire();

    if (numLinuxDudes == 2) {
```

```

        /* Fix: numLinuxDudes -= 2; */

        linuxwait->Signal();

        linuxwait->Signal();

        RowBoat();
    }

    else if (numMSDudes == 1 && numLinuxDudes == 1) {

        /* Fix: numMSDudes--;numLinuxDudes--; */

        mswait->Signal();

        linuxwait->Signal();

        RowBoat();
    }

    else {

        numLinuxDudes++;

        linuxwait->wait(mutex);

        numLinuxDudes--; /* Remove this line to fix */

    }

    mutex->Release();
}

void MSDudeArrives () {

    mutex->Acquire();

    if (numMSDudes == 2) {

```



```

        /* Fix: numMSDudes -= 2; */

        mswait->Signal();

        mswait->Signal();

        RowBoat();

    }

    else if (numLinuxDudes == 2) {

        /* Fix: numLinuxDudes -= 2; */

        linuxwait->Signal();

        linuxwait->Signal();

        RowBoat();

    }

    else {

        numMSDudes++;

        mswait->wait(mutex);

        numMSDudes--; /* Fix: remove this line */

    }

    mutex->Release();

}

```

The above solution is not correct. The main problem with this solution is that the number of MS or Linux dudes leaving the shore gets updated after the dudes are signalled to enter the boat. This introduces some interesting races. Here is an example. Suppose 2 Linux dudes are there and a third one (name him Linux3) comes along. Linux3 sees `numLinuxDudes == 2` and is about to signal the other 2 Linux dudes. If Linux3 gets preempted before it signals there is a chance another Linux dude (name him Linux4) arrives. Linux4 will not enter the critical section and will not update the shared counter, but if the mutex is implemented with a queue, he will be at the head of the mutexe's

queue. When the system switches back to Linux3, Linux3 signals Linux1 and Linux2 and calls RowBoat. But consider what happens to Linux1 and Linux2. They will both go back to the ready queue, but according to the semantics of condition variables, they must reacquire the mutex, and this will happen later than Linux4. Eventually Linux3 will release the mutex, and Linux4 will get in (Linux1 and Linux2 are behind him in the mutex's queue). Linux 4 see numLinuxDudes == 2 (remember that Linux1 and Linux2 are still inside the wait trying acquiring the mutex), he sends two useless signals and mistakenly calls RowBoat, although he is the only guy the shore. A similar scenario occurs if instead of Linux4, Microsoft1 arrives. Microsoft1 will too send two useless signals and call RowBoat without enough people on the boat. All this happens just because the number of Linux/Microsoft dudes are not updated before the members of a boat load are signaled but after. To fix this code, you need to update numLinuxDudes and numMSDudes right before a Linux/MS dude gets signaled to enter the boat, as shown with red lines.

The solution has another problem; it does not always prevent unnecessary waiting. An example that demonstrates where it causes waiting when a boat could be sent is the following order of arrivals:

```
MSDudeArrives
MSDudeArrives
LinuxDudeArrives
LinuxDudeArrive
```

If this is the order of arrivals, then two Linux Dudes should be sent across the river with one MS Dude, but that will not happen. LinuxDudeArrives() only checks that (numMSDudes == 1) in its second if statement, when it should really be checking that (numMSDudes >= 1) because as long as there are any MS Dudes at this point, then a boat can be sent, not just exactly one MS Dude.

Write a solution to the dining philosophers using locks and condition variables. Your solution must prevent philosopher starvation.

```
/*
 * Each philosopher is a thread that does the following:
 */

philosopherMain(int myId, Table *table){

    while(1){

        table->startEating(myId);
```

```
        eat();

        table->doneEating(myId);
    }
}
```

```
typedef enum stick_status{STICK_FREE, STICK_USED}
stick_status;
```

```
const static int NOT_WAITING = -999;
```

```
public class Table{

    public:

    Table(int nSeats);

    void startEating(int philosopherId);
    void doneEating(int philosopherId);

    private:

    mutex lock;

    cond cv;

    int nseats;

    stick_status stickStatus[];

    int entryTime[];

    int currentTime;
```

```

    int okToGo(int id);
}

Table::Table(int ns)
{
    int ii;

    nseats = ns;

    stickStatus = (stick_status *)malloc(nseats *
sizeof(stick_status));

    entryTime = (int *)malloc(nseats * sizeof(int));

    for(ii = 0; ii < nseats; ii++){
        stickStatus[ii] = STICK_FREE;

        entryTime[ii] = NOT_WAITING;
    }

    currentTime = 0;
}

void
Table::startEating(int id)
{
    lock.acquire();

    entryTime[id] = currentTime;

    currentTime++;

    while(!okToGo(id)){

```

```
        cv.wait(&lock);

    }

    stickStatus[stickLeft(id)] = STICK_USED;
    stickStatus[stickRight(id)] = STICK_USED;
    entryTime[id] = NOT_WAITING;
    lock.release();
}
```

```
void
Table::doneEating(int id)
{
    lock.acquire();

    stickStatus[stickLeft(id)] = STICK_FREE;
    stickStatus[stickRight(id)] = STICK_FREE;
    cv.broadcast(&lock);
    lock.release();
}
```

```
int
Table::okToGo(int id)
{
    assert(lock is held on entry);

    /*
```

```

    * OK to go if both left and right sticks
    * are free AND for each stick my neighbor
    * is not waiting, or my neighbors waiting
    * number is larger than mine

    */

    if(stickStatus[stickLeft(id)] != STICK_FREE){

        return 0;

    }

    if(stickStatus[stickRight(id)] != STICK_FREE){

        return 0;

    }

    if(entryTime[seatRight(id)] != NOT_WAITING

        &&

        entryTime[seatRight(id)] < entryTime[id]){

        return 0;

    }

    if(entryTime[seatLeft(id)] != NOT_WAITING

        &&

        entryTime[seatLeft(id)] < entryTime[id]){

        return 0;

    }

    return 1;

}

```

The following is an implementation of a stack data structure for a multithreaded application. Identify the bugs, if any, and correct them.

```
#include "Exception.h"

#include "Semaphore.h"

#include

const MaxStackSize = 100;

class Stack

// throws an exception object when popping an empty stack,
// and when pushing

// into a full stack

{

    private:

        int s[MaxStackSize];

        int stackp; // stack pointer

        Exception * e; // For error handling

        Semaphore * sem; // For mutual exclusion

    public:

        Stack();

        ~Stack(){};

        int Pop(void);

        void Push(int item);

};
```

```

Stack::Stack( )
{
    stackp = MaxStackSize;
    e = new Exception();
    sem = new Semaphore(1);
}

```

```

int Stack::Pop(void)
{
    P(sem)

    if(stackp == MaxStackSize){
        e->SetErrorMsg("Popping empty stack");
        e->SetErrorLocation("Stack::Pop()");
        throw(e);

```

Error: Before throwing the exception, we must release the lock (i.e. V(sem)), or the stack object will not be accessible to any process any time in the future.

```

    }

    V(sem);

    return s[stackp++];

```

Error: We are incrementing stackp after releasing the lock!!


```
}
```

```
void Stack::Push(int item)
```

```
{
```

```
    P(sem)
```

```
    if(stackp == 0)
```

```
    {
```

```
        e->SetErrorMsg("Pushing to a full stack");
```

```
        e->SetErrorLocation("Stack::Push()");
```

```
        throw(e);
```

Error: Before throwing the exception, we must release the lock (i.e. V(sem)), or the stack object will not be accessible to any process any time in the future.

```
    }
```

```
    s[--stackp] = item;
```

```
    V(sem);
```

```
}
```

Consider the following program fragment:

```
P(s1);
```

```
a++;
```

```
P(s2);
```

```
v++;
```

```
V(s2);
```

```
V(s1);
```

(s1, s2 are semaphores). All variables are automatic, meaning that each thread gets its own private copy of a and v from its stack. Now, consider two threads running this fragment of code simultaneously, can there be a deadlock? Why, or why not?

Both semaphores will be acquired in the same order by both threads. Since this is a linear order, there can be no situation where a deadlock could occur.

Consider the following program fragment:

```
if(a > 0)
```

```
    P(s1);
```

```
else
```

```
    P(s2);
```

```
b++;
```

```
P(s3);
```

```
if(b < 0 && a <= 0)
```

```
    P(s1);
```

```
else if(b >= 0 && a > 0)
```

```
    P(s2);
```

```
else
```

```
    P(s4);
```

```
a++;
```

```
V(s4);
```

```
V(s3);
```

```
V(s2);
```

```
V(s1);
```

s1, s2, s3 and s4 are semaphores. All variables are automatic, that is each thread as a local copy of a and b that it modifies. Now, consider two threads running this fragment of code simultaneously, can there be a deadlock? Why, or why not?

Yes, there can be a deadlock. Consider the scenario where thread 1 starts by locking semaphore s1 then gets switched out while thread 2 runs. Thread 2 may start running with variable a set to 0 (remember, because the variables are automatic, each thread has its own independent set). Thread 2 will acquire semaphore s2, then proceeds to acquire s3. Then, if (b<0 && a <=0), it will try to acquire s1. This forces thread 2 to wait for thread 1 which holds this semaphore. Now, thread 1 may proceed but will block when it tries to acquire semaphore s3. This forms a cyclic wait, and a deadlock occurs.

Solve problems 7.8 and 7.9 from the textbook.

7.8 The Sleeping-Barber Problem. A barbershop consists of a waiting room with n chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

```
customers=0;
```

```
barbers=1;
```

```
mutex=1;
```

```
waiting=0;
```

```
void barber(void)
```

```
{
```

```
    while(true){
```

```
        wait(customers);
```

```

        wait(mutex);

        waiting=waiting-1;

        signal(barbers);

        signal(mutex);

        cut_hair();

    }

}

void customer(void)
{
    wait(mutex);

    if(waiting < NUMBER_OF_CHAIRS){

        waiting ++;

        signal(customers);

        signal(mutex);

        wait(barbers);

        get_haircut();

    }else{

        signal(mutex);

    }

}

```

7.9 The Cigarette-Smokers Problem. Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers.

```
Tobacco = 0;
```

```
Match = 0;
```

```
Paper = 0;
```

```
void agent()
```

```
{
```

```
    lock.acquire();
```

```
    while(1){
```

```
        while (!((Tobacco == 0) && (Match == 0) && (Paper == 0))) {
```

```
            IngredientConsumed.wait(&lock);
```

```
        }
```

```
        chooseIngredients(&Tobacco, &Match, &Paper); // Sets two of the variables to 1.
```

```
        IngredientReady.broadcast(&lock);
```

```
    }
```

```
    lock.release();
```

```
}
```

```

void matchSmoker()
{
    lock.acquire();

    while(1){

        while( !( (Tobacco == 1) && (Paper == 1) ) ){

            IngredientReady.wait(&lock);

        }

        //      smoking;

        Tobacco = 0;

        Paper = 0;

        IngredientConsumed.signal(&lock);

    }

    lock.release();
}

```

```

void paperSmoker()
{
    lock.acquire();

    while(1){

        while( !( (Tobacco == 1) && (Match == 1) ) ){

            IngredientReady.wait(&lock);

        }

    }
}

```

```

        //      smoking;

        Tobacco = 0;

        Match = 0;

        IngredientConsumed.signal(&lock);

    }

    lock.release();
}

void tobaccoSmoker()
{
    lock.acquire();

    while(1){

        while( !( (Match == 1) && (Paper == 1) ) ){

            IngredientReady.wait(&lock);

        }

        //      smoking;

        Match = 0;

        Paper = 0;

        IngredientConsumed.signal(&lock);

    }

    lock.release();
}

```

3 Solutions

8.2 Is it possible to have a deadlock involving only one single process? Explain your answer.

No, because of the hold-and-wait condition. If you are holding resources, you are not waiting for them.

8.4 Consider the traffic deadlock depicted in the Figure 8.8

1. Show that the four necessary conditions for deadlock indeed hold in this example.
2. State a simple rule that will avoid deadlocks in this system.

a. The four necessary conditions for deadlock hold in this example for the following reasons:

- (i) Mutual Exclusion: Each of the vehicles presents in the streets hold a non-sharable resource: the part of the road they occupy, which they cannot share with the other vehicles.
- (ii) Hold and Wait: Each of the vehicles hold the space resource they occupy and are waiting the space in front of them to be freed by other waiting vehicles.
- (iii) No Preemption: There is no possibility of preemption as none of the vehicles can give up their resource. In this situation preemption would have to take the form of a vehicle pulling into a parking lot, or a crane reaching down and lifting a vehicle off the road.
- (iv) Circular Wait: Circular wait is present because each vehicle is waiting for the space in front of it, and some vehicles occupy spaces where two vehicles wait on them. It is thus possible to trace a cycle of waiting cars. This is the weakest assertion in the set, though, and is clearly untrue out at the edges somewhere, since some car can clearly move someplace in the city. If you have ever experienced grid-lock, though you know that this is small comfort, and that a rule to avoid even "local" deadlock is extremely desirable.

b. The simple rule that could be used to avoid traffic deadlocks in such a system is that intersections should always remain clear as lights change. In this way, the resource of space in the intersection is freed for use at periodic intervals (light changes).

8.7 Prove that the safety algorithm presented in Section 8.5.3 requires an order of $m \cdot n^2$ operations.

We'll prove the complexity of this algorithm by looking at each step in turn.

- 1) In step one, the two arrays Work and Finish are initialized which take times $O(m)$ and $O(n)$ respectively. This yields a complexity of step one of $O(m+n)$.
- 2) In step two, we look at each element of Finish from 1 to n . For every element of Finish which is true we may look at upto n elements of the corresponding vector in the Need

array to check that each element is less than or equal to the elements of the Work vector. This would look like a double for loop, both going from 1 to n. This step requires a time of $O(n^2)$.

3) For each iteration of the inner loop from step two we might call step three. In step three we update the Work array which takes time m. The two other lines of this step can be ignored as constants. This step on its own requires time $O(m)$. Combined with the previous step, which would happen in the worst case, the total time for steps two and three combined is $O(m \cdot n^2)$. The worst case I mention here is when all elements of Finish are FALSE and Need[i] is less than Work for all processes. Obviously, this step requires constant time: $O(1)$.

When put together, these steps require a combined:
 $O(m+n) + O(m \cdot n^2) + O(1)$
which can be reduced to:
 $O(m \cdot n^2)$.

8.8 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and, therefore it will return its resources when done.

8.9 Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock-free if the following two conditions hold:

- a. the maximum need of each process is between 1 and m resources
- b. The sum of all maximum needs is less than $m + n$

A) $(\text{SUM from } i = 1 \text{ to } N) \text{ Max}[i] < m + n$

B) $\text{Max}[i] \geq 1$ for all i

Proof: $\text{Need}[i] = \text{Max}[i] - \text{Allocation}[i]$

If there exists a deadlock state then:

C) $(\text{SUM from } i = 1 \text{ to } N) \text{ Allocation}[i] = m$

Use A to get: $(\text{SUM}) \text{ Need}[i] + (\text{SUM}) \text{ Allocation}[i] = (\text{SUM}) \text{ Max}[i] < m + n$

Use C to get: $(\text{SUM}) \text{ Need}[i] + m < m + n$

Rewrite to get: $(\text{SUM from } i = 1 \text{ to } n) \text{ Need}[i] < n$

This implies that there exists a process $P[i]$ such that

$\text{Need}[i]=0$. Since $\text{Max}[i] \geq 1$ it follows that $P[i]$ has atleast one

Resource that it can release. Hence the system cannot be in a deadlock

State.

8.12 Can a system detect that some of its processes are starving? If you answer yes, explain how it can. If you answer no, explain how the system can deal with the starvation problem.

No. A process starves if it never gets to run. If a given process has not run for 1 second, has it starved? How about if it has not run for 10 seconds? A minute? An hour? None of these indicate that a process is starving, since it may get to run in the next second. However, as the amount of time that a process has waited gets longer and longer, the probability that it is starving goes up.

This argument depends on the fact that no numeric criteria exist for declaring a process to have starved. If, on the other hand, we declared a process to have starved if it when 10 seconds without running, then we might be able to answer "yes".

8.13 Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	

P4 0 0 1 4 0 6 5 6

Answer the following questions using the banker's algorithm:

(a) what is the content of the matrix Need?

(b) Is the system in a safe state?

(c) If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

(a) Need = Max - Allocation

Need

A B C D

P0 0 0 0 0

P1 0 7 5 0

P2 1 0 0 2

P3 0 0 2 0

P4 0 6 4 2

(b) The system is in a safe state. For example: P0, P1, P3, P4, is a safe sequence.

(c) After satisfying P1 request, the system becomes the following state.

Allocation Max Need Available

A B C D A B C D A B C D A B C D

P0 0 0 1 2 0 0 1 2 0 0 0 0 1 1 0 0

P1 1 4 2 0 1 7 5 0 0 3 3 0

P2 1 3 5 4 2 3 5 6 1 0 0 2

P3 0 6 3 2 0 6 5 2 0 0 2 0

P4 0 0 1 4 0 6 5 6 0 6 4 2

By using the safety algorithm, the system is still in a safe state and P0, P2, P1, P3, P4 is a safe sequence.

8.14 Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector Available initialized to (4,2,2). If process P0 ask for (2,2,1), it gets them. If P1 asks for (1,0,1), it gets them. Then, if P0 asks for (0,0,1), it is blocked (resource not available). If P2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P0 since P0 is blocked). P0s Allocation vector goes down to (1,2,1), and its Need vector goes up to (1,0,1).

a. Can deadlock occur? If so, give an example. If not, which necessary condition cannot occur?

b. Can indefinite blocking occur?

a. Deadlock cannot occur because preemption exists.

b. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as what happens to P0 in the above case.

Part 1: Theory

Most round-robin schedulers use a fixed size quantum. Give an argument in favor of and against a small quantum. Give an argument in favor of and against a large quantum. An argument against a small time quantum: Efficiency. A small time quantum requires the timer to generate interrupts with short intervals. Each interrupt causes a context switch, so overhead increases with a larger number of interrupts. An argument for a small time quantum: Response time. A large time quantum will reduce the overhead of context switching since interrupts will be generated with relatively long intervals, hence there will be fewer interrupts. However, a short job will have to wait longer time on the ready queue before it can get to execute on the processor. With a short time quantum, such a short job will finish quicker and produces the result to the end user faster than with a longer time quantum.

Many scheduling algorithms are parameterized. For instance, the round-robin algorithm requires a parameter to indicate the time quantum. The multi-level feedback (MLF) scheduling algorithm requires parameters to define the number of queues, the scheduling algorithm for each queue, and the criteria to move processes between queues (and perhaps others. . .). Hence, each of these algorithms represents a set of algorithms (e.g., the set of round-robin algorithms with different quantum sizes). Further, one set of

algorithms may simulate another (e.g., round-robin with infinite quantum duration is the same as first-come, first-served (FCFS)). For each of the following pairs of algorithms, answer the following questions:

Priority scheduling and shortest job first (SJF)

State the parameters and behavior of priority scheduling

Parameter: Each job has a priority P

Behavior: Choose the job with the lowest numerical priority

State the parameters and behavior of SJF

Parameter: Each job has a remaining time T

Behavior: Choose the job with the lowest remaining T

Can SJF simulate priority scheduling for all possible parameters of priority scheduling? (How or why not: State how to set SJF scheduling parameters as a function of priority scheduling parameters or explain why this cannot be done.)

2 possible answers: (1) No - although both schedulers are priority queues, SJF is more restrictive than a general priority scheduler since a job's time may not correspond to its priority. (2) Yes - "spoof" the scheduler by passing priority in place of time.

Can priority scheduling simulate SJF for all possible parameters of SJF? (How or why not.)

Yes - set the priority P to the remaining time T

Multilevel feedback queues and first come first served (FCFS)

State the parameters and behavior of multi-level feedback queues

Parameters: N (# queues), scheduling algorithm for each queue, function that selects in which queue to place a job, criteria to interrupt a running job

Behavior: Always schedule the job from the head of the lowest #'d non-empty queue according to that queue's scheduling algorithm; interrupt a running job when told to by the interrupt criteria; place newly arrived jobs or interrupted jobs in a queue based on the selection function

State the parameters and behavior of FCFS

No parameters

Behavior: Place arriving jobs at the back of a FIFO queue; when scheduling a new job, choose the one from the front of the FIFO queue; never interrupt a running job

Can FCFS simulate multi-level feedback for all possible parameters of multi-level feedback? (How or why not?)

No. Counterexample: make the top queue of MLF run RR with a short time quantum. If two jobs arrive and each is longer than the quantum, the MLF scheduler will allow both jobs to run for at least one quantum before either completes, but FCFS can never interrupt a running job, so one of the two will complete before the other has a chance to run.

Can multi-level feedback scheduling simulate FCFS for all possible parameters of FCFS? (How or why not?)

Yes. Make MLF have 1 queue and have that one queue run FCFS. Function to select which queue - always choose that queue. Policy to interrupt - Never.

Priority scheduling and first come first served (FCFS)

Can FCFS simulate priority scheduling for all possible parameters of priority scheduling? (How or why not?)

No. Counterexample: Suppose job 1 arrives at time 1 with priority 10 and length 1,000,000 and job 2 arrives at time 2 with priority 1 and length 1,000,000. FCFS will run job 1 to completion, then job 2 to completion. Priority will run job 1 for 1 unit, then job 2 to completion, then job 1 to completion.

Can priority scheduling simulate FCFS for all possible parameters of FCFS? (How or why not?)

Yes. Set all jobs to equal priority (we assume that the priority queue breaks ties with FCFS and infinite time quantum)

Round-robin and shortest job first (SJF)

State the parameters and behavior of round robin

Parameter: Quantum Q

Behavior: Run FIFO queue, but interrupt a running job Q units after it begins running and move it to the end of the FIFO queue

Can round robin simulate SJF for all possible parameters of SJF? (How or why not?)

No. Counterexample. 3 jobs arrive: job 1 at time 0 is 10 seconds long, job 2 arrives at time 1 and is 100 seconds long, job 3 at time 2 is 10 seconds long. SJF must run 1 to completion, then 3 to completion, and then 2 to completion. Claim: RR must run at least one quantum of job 2 before running any job 3.

Can SJF simulate round robin for all possible parameters of round robin? (How or why not?)

No, same counterexample

Part 2: Problems

Level of difficulty is high on this one. As a system administrator you have noticed that usage peaks between 10:00AM to 5:00PM and between 7:00PM to 10:00PM. The company's CEO decided to call on you to design a system where during these peak hours there will be three levels of users. Users in level 1 are to enjoy better response time than users in level 2, who in turn will enjoy better response time than users in level 3. You are to design such a system so that all users will still get some progress, but with the indicated preferences in place.

Will a fixed priority scheme with pre-emption and 3 fixed priorities work? Why, or why not?

No, it will not work. A fixed priority scheme can cause starvation. The required solution should enable all users to make progress. Fixed priority does not guarantee progress for processes with low priorities.

Will a UNIX-style multi-feedback queue work? Why, or why not?

No, it will not work. The multi-feedback queuing system will cause processes in level 1 to get less time on the CPU if they stay in the system for very long. So, even though a level-1 process may start at the highest level in the feedback queue, its priority will degrade over time. So this solution does not satisfy the requirements.

If none of the above works, could you think of a scheduling scheme that meets the requirements?

A process of level-1 will have 3 entries in the ready queue, distributed evenly over the queue. A process of level-2 will have 2 entries, while a process of level 3 will have one entry. In a run, a process at level 1 will get 3 times as much as a process at level 3 on the CPU. Care should be taken such that when a process requests an I/O operation, that all its entries would be removed from the ready queue simultaneously. Also, when a process is added to the ready queue, it has to be entered with all its entries even distributed over the entire queue. Other solutions such as a lottery scheduler are possible.

A periodic task is one that is characterized with a period T , and a CPU time C , such that for every period T of real time, the task executes C time on the CPU. Periodic tasks occur in real-time applications such as multimedia. For example, a video display process needs to draw a picture every 40 msec (outside the US, Japan and Canada). To do so, such a process will have a period of 40 msec, and within each, it will require some time to do the data copying and drawing the picture after getting it from the video stream.

Admission control is a policy implemented by the process manager in which the manager decides whether it will be able to run a task with given parameters and with the existing load. Of course, if the process manager over commits itself, a periodic task will not be able to meet its processing requirement every period. For the video display process example, the picture flow will slow down and a viewer will notice and get irritated.

Assume that a system has 5 processes, an editor, a mail checking program, a video display process, a network browser and a clock display. The requirements are such that the mail checking program has to execute for 10 msec every 1sec, the video display must execute 25 msec every 40 msec, and the clock display must execute for 10 msec every 1 sec.

What scheduling policy would you use for such a system?

Two process schedulers are available for you. The first would consume about 5 msec of overhead in context switching every 40 msec, while the second consumes 0.5 msec of overhead in context switching every 40 msec. The first ensures that the periodic processes will get time on the CPU every 40 msec if they are ready, while the second uses simple round-robin scheduling. Which one is better suited for this application batch? Which one is more efficient?

Assume that the clock display program actually underestimated the time it wanted to run, and instead has now to run for 20 msec every 40 msec. What should the system do? How could it do it? Explain your answers. If an audio unit is to be installed with an audio playback processing requiring 15 msec every 40 msec, would you be able to admit the process into the system? If yes explain why, and if not, explain why and suggest what the system could do.

Solution:

1. A fixed priority scheme in which the video display would have the highest priority, followed by the clock, then followed by the rest.
2. While the second scheduling algorithm is more efficient than the first, it is practically useless for such a batch of applications because simple round-robin does not give the necessary guarantees that the video display program will run every 40- msec.
3. The system has to be able to stop the offending process. To do so, the system must set timers that would generate interrupts at the time limit by which a task has to finish. So, if the task has not finished after its allotted time has expired, then it is simply switched out and another application is run. This requires a very complex implementation to ensure that the timers are being handled properly.
4. The system cannot admit such a process, because it does not have the capacity to handle the workload and guarantee the time limits required for the other processes. In situations like this, the system may simply tell the user "sorry", or it may give her the ability to stop some of the existing applications so that the freed capacity be used to run the new application. Or, it may simply tell the user "we will not run this as it should, do you want to continue anyway?".

Comparison of FIFO, RR and SRTF Algorithms. Given the following mix of job, job lengths, and arrival times, assume a time slice of 10 and compute the completion for each job and average response time for the FIFO, RR, and SRTF algorithms. Please use the following table format for your solution.

Solution:

			Scheduling Algorithms					
Job	Length (secs)	Arrival time	FIFO		RR		SRTF	
			Completion time	Response time	Completion time	Response time	Completion time	Response time
0	85	0	85	85	220	220	220	220
1	30	10	115	105	80	70	40	30
2	35	10	150	140	125	115	75	65
3	20	80	170	90	145	65	100	20
4	50	85	220	135	215	130	150	65
Average Response Time			111		120		80	

Note:

RR's scheduling is:

```

0: 0
10: 1    * if scheduling 0, the result is listed below.
20: 2
30: 0
40: 1
50: 2
60: 0
70: 1
80: 2    ----->Job1 completes
90: 0
100: 3
110: 4
120: 2
125: 0    ----->Job2 completes
135: 3
145: 4    ----->Job3 completes
155: 0
165: 4
175: 0
185: 4
195: 0

```

205: 4
 215: 0 ----->Job4 completes
 220: ----->Job0 completes

RR can also be:

			Scheduling Algorithms					
Job	Length (secs)	Arrival time	FIFO		RR		SRTF	
			Completion time	Response time	Completion time	Response time	Completion time	Response time
0	85	0	85	85	220	220	220	220
1	30	10	115	105	90	80	40	30
2	35	10	150	140	135	125	75	65
3	20	80	170	90	155	75	100	20
4	50	85	220	135	220	135	150	65
Average Response Time			111		125		80	

Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for CPU (in the ready queue, but not running), its priority changes at a rate of a (i.e., $P(t) = P_0 + a * (t - t_0)$ where t_0 is the time at which the process joins the ready queue). Similarly, when it is running, its priority changes at a rate b . All processes are given a priority 0 when they enter the ready queue. The parameters a and b can be set to obtain many different scheduling algorithms.

What is the algorithm that results from $b > a > 0$?

What is the algorithm that results from $a < b < 0$?

Solution:

FCFS (First-come, first-served). All the processes in the ready queue has the same initial priority 0 ($P_0=0$). Their priority increases at the same rate $a(a>0)$. Thus, the earlier the process enters the ready queue (t_0), the higher its priority will be ($P(t) = a * (t - t_0)$). Once the process with the highest priority (first-come compared to other process in the ready queue) is running, its priority increases at a higher rate ($b>a>0$) than the priorities of

those processes in the ready queue. So no other process will preempt it and it will run to its completion.

LCFS (Last-come, first-served). All the processes in the ready queue has the same initial priority 0 ($P_0=0$). Their priority decreases at the same rate $a(a<0)$. Thus, the earlier the process enters the ready queue (t_0), the lower its priority will be ($P(t) = a * (t - t_0)$). Once the process with the highest priority (last-come compared to other process in the ready queue) is running, no other process in the ready queue will preempt it since its priority decreases at a lower rate ($a < b < 0$) than the priorities of those processes in the ready queue. But it will be preempted by any new coming process because the new process's priority is 0 which is the highest possible priority.

5 Solutions

Part 1: Theory

Describe the following contiguous allocation algorithms: first-fit, best-fit, worst-fit.

Solution:

First-Fit: Search the list of available memory and allocate the first block that is big enough.

Best-Fit: Search the entire list of available memory and allocate the smallest block that is big enough.

Worst-Fit: Search the entire list of available memory and allocate the largest block.

How is protection enforced in a paging system?

Solution:

Protection is enforced in a paging system in two different ways. First, access-mode bits limit the types of accesses that are allowed to a particular frame. For example a frame may be marked as read-only in which case any attempt to write to this frame would cause a hardware trap to the operating system. Second, the hardware automatically checks the logical address of the frame and makes sure that it is within the appropriate range. For example, an access to logical address 12,000 in an address space with a highest address of 10,468 would be illegal. This is the opposite case of trying to access memory with a negative address, clearly also illegal. The fact that the physical addresses associated with each logical address are not necessarily contiguous is not important to the security of the paging system.

What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how you could use this effect to decrease the amount of time needed to copy a large amount of memory from one place to another. What would the effect of updating some byte in one shared page be on the other shared page?

Solution:

By allowing two entries in a page table to point to the same page frame in memory, users

can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. "Copying" large amounts of memory could be effected by having different page tables point to the same memory location.

However, sharing of non-reentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user's copy.

Name two advantages of segmentation over paging. Why are the two schemes combined in some systems?

Solution:

- 1) Speed. Reloading segment registers to change address spaces is much faster than switching page tables.
- 2) Segment descriptor tables consume less memory than page tables.
- 3) x86 page table entries do not have an 'Executable' bit. With segmentation, you can make a region of memory executable (code) or not (data).
- 4) Memory allocation unit is a logically natural view of program.
- 5) Segments can be loaded individually on demand (dynamic linking).
- 6) Natural unit for protection purposes.
- 7) No internal fragmentation.

Segmentation and paging are sometimes combined in order to improve upon each other. Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single segment table entry with a page table address of zero. Paged segmentation handles the case of having very long segments that require a lot of time for allocation. By paging the segments, we reduce wasted memory due to external fragmentation as well as simplify the allocation.

Define external and internal fragmentation and identify the differences between them.

Solution:

Internal fragmentation is where the memory manager allocates more for each allocation than is actually requested. Internal fragmentation is the wasted (unused) space within a page. For example if I need 1K of memory, but the page size is 4K, then there is 3K of wasted space due to internal fragmentation.

External fragmentation is the inability to use memory because free memory is divided into many small blocks. If live objects are scattered, the free blocks cannot be coalesced, and hence no large blocks can be allocated. External fragmentation is the wasted space outside of any group of allocated pages that is too small to be used to satisfy another request. For example if best-fit memory management is used, then very small areas of memory are likely to remain, which may not be usable by any future request.

Both types of fragmentation result in free memory that is unusable by the system.

Part 2: Problems

Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how would each of the first-fit, best-fit and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB and 426 KB (in that order) ? Which algorithm makes the most efficient use of memory?

Solution:

First-Fit:

212K is put in 500K partition.

417K is put in 600K partition.

112K is put in 288K partition (new partition $288K = 500K - 212K$).

426K must wait.

Best-Fit:

212K is put in 300K partition.

417K is put in 500K partition.

112K is put in 200K partition.

426K is put in 600K partition.

Worst-Fit:

212K is put in 600K partition.

417K is put in 500K partition.

112K is put in 388K partition.

426K must wait.

In this example, Best-Fit turns out to be the best.

Consider a virtual address space of eight pages with 1024 bytes each, mapped onto a physical memory of 32 frames. How many bits are used in the virtual address ? How many bits are used in the physical address ?

Solution:

There are 13 bits in the virtual address.

There are 15 bits in the physical address.

Consider a paging system with the page table stored in memory. If a memory reference takes 200 nanoseconds how long does a paged memory reference take ? If we add a TLB and 75 percent of all page-table references are TLB hits, what is the effective memory reference time? (Assume that finding a page-table entry in the TLB takes zero time, if the entry is there.)

Solution:

A paged memory reference would take 400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

The effective memory access time is:

$$\text{E.A.T.} = 0.75 * (200 \text{ nanoseconds}) + 0.25 * (400 \text{ nanoseconds}) = 250 \text{ nanoseconds}$$

Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

- a. 0, 430
- b. 1, 10
- c. 2, 500
- d. 3, 400
- e. 4, 112

Solution:

- a. $219 + 430 = 649$

- b. $2300 + 10 = 2310$
- c. illegal reference, trap to operating system
- d. $1327 + 400 = 1727$
- e. illegal reference, trap to operating system

Devise a scenario showing that Best-fit allocation outperforms all alternatives discussed in class. Repeat for First-fit, Worst-fit, and Buddy Allocation.

Solution:

(x) A yK - means: in step x we allocate yK of memory

(x) D (y) - means: in step x we deallocate the memory allocated in step y

Best fit outperforms:

(1) A 5K

(2) A 8K

(3) A 3K - buddy cannot do this

(4) D (1)

(5) D (3)

(6) A 3K - first and worst take the 5K part, best the 3K part

(7) A 5K - first and worst cannot do this, best can

Worst fit outperforms:

(1) A 3K

(2) A 8K

(3) A 5K - buddy cannot do this

(4) D (1)

(5) D (3)

(6) A 2K - first and best take the 3K part, worst the 5K part

(7) A 3K - first and best take the 5K part, worst a 3K

(8) A 3K - first and best cannot do this, worst can

First fit outperforms:

(1) A 4K

(2) A 2K

(3) A 2K

(4) A 3K

(5) A 5K - buddy cannot do this

(6) D (1)

(7) D (3)

(8) D (5)

(9) A 1K - best takes the 2K part, worst the 5K part, first the 4K part

(10) A 3K - best takes the 4K part, worst a 4K part, first the 3K part

(11) A 2K - best takes the 5K part, worst the 4K part, first the 2K part

(12) A 5K - best and worst cannot do this, first can

Buddy outperforms:

(1) A 2K

(2) A 4K

(3) A 8K

(4) D (1) - only buddy can merge the 2K with the neighbouring 2K to a 4K part

(5) A 4K - best, worst and first cannot do this, buddy can

A minicomputer uses the buddy system for memory management. Initially it has one block of 256K at address 0. After successive requests of 7K, 26K, 34K and 19K come in, how many blocks are left and what are their sizes and addresses?

Solution:

7K: We recursively break the address space into 2 halves until we have:

8K - 8K - 16K - 32K - 64K - 128 K

The first segment is used to satisfy the 7K request.

26K: We use the 32K block to allocate the request.

8K - 8K - 16K - 32K - 64K - 128 K

34K: We use the 64K block to satisfy the request:

8K - 8K - 16K - 32K - 64K - 128 K

19K: Since 8K and 16K cannot satisfy the request on their own, we need to break the big 128K block recursively until we get the size we need. The blocks will look like:

8K - 8K - 16K - 32K - 64K - 32K - 32K - 64K

This question refers to an architecture using segmentation with paging. In this architecture, the 32-bit virtual address is divided into fields as follows:

4 bit segment number	12 bit page number	16 bit offset
----------------------	--------------------	---------------

Here are the relevant tables (all values in hexadecimal):

Segment Table		Page Table A		Page Table B	
0	Page Table A	0	CAFE	0	F000
1	Page Table B	1	DEAD	1	D8BF
x	(rest invalid)	2	BEEF	x	(rest invalid)
		3	BA11		
		x	(rest invalid)		

Find the physical address corresponding to each of the following virtual addresses (answer "bad virtual address" if the virtual address is invalid):

00000000

20022002

10015555

Solution:

CAFE0000

bad virtual address

D8BF5555

6 Solutions

10.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

Solution:

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

10.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page reference string has length p ; n distinct page numbers occur in it.

Answer these questions for any page-replacement algorithms:

- What is a lower bound on the number of page faults?
- What is an upper bound on the number of page faults?

Solution:

- n
- p

10.3 A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

Solution:

The virtual address in binary is:

0001 0001 0001 0010 0011 0100 0101 0110

Since the page size is 2^{12} , the page table size is 2^{20} . Therefore the low-order 12 bits "0100 0101 0110" are used as the displacement into the page, while the remaining 20 bits "0001 0001 0001 0010 0011" are used as the displacement in the page table.

10.4 Which of the following programming techniques and structures are "good" for a demand-paged environment? Which are "bad"? Explain your answers.

- a. Stack
- b. Hashed symbol table
- c. Sequential search
- d. Binary search
- e. Pure code
- f. Vector operations
- g. Indirection

Solution:

- a. Stack -- good.
- b. Hashed symbol table -- not good.
- c. Sequential search -- good.
- d. Binary search -- not good.
- e. Pure code -- good.
- f. Vector operations -- good.
- g. Indirection -- not good.

10.5 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

Solution:

$$0.2 \text{ microsec} = (1 - P) * 0.1 \text{ microsec} + (0.3P) * 8 \text{ millisec} + (0.7P) * 20 \text{ millisec}$$

$$0.1 = -0.1P + 2400P + 14000P$$

$$0.1 \approx 16,400P$$

$$P \approx 0.000006$$

10.9 Consider a demand-paging system with the following time-measured utilizations:

CPU utilization 20%

Paging disk 97.7%

Other I/O dev. 5%

For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.

- a. Install a faster CPU.
- b. Install a bigger paging disk.
- c. Increase the degree of multiprogramming.
- d. Decrease the degree of multiprogramming.
- e. Install more memory.
- f. Install a faster hard disk, or multiple controllers with multiple hard disks.
- g. Add prepaging to the page-fetch algorithms.
- h. Increase the page size.

Solution:

The system is obviously spending most of its time paging, indicating over-allocation of memory. If the level of multiprogramming is reduced resident processes would page fault less frequently and the CPU utilization would improve. Another way to improve performance would be to get more physical memory or a faster paging drum.

- a. Install a faster CPU -- No.
- b. Install a bigger paging disk -- No.
- c. Increase the degree of multiprogramming -- No.
- d. Decrease the degree of multiprogramming -- Yes.
- e. Install more memory -- Likely to improve CPU utilization as more pages can remain resident and not require paging to or from the disks.
- f. Install a faster hard disk, or multiple controllers with multiple hard disks -- Also an improvement, for as the disk bottleneck is removed by faster response and more throughput to the disks, the CPU will get more data more quickly.
- g. Add prepaging to the page-fetch algorithms -- Again, the CPU will get more data faster, so it will be more in use. This is only the case if the paging action is amenable to prefetching (i.e., some of the access is sequential).

h. Increase the page size -- Increasing the page size will result in fewer page faults if data is being accessed sequentially. If data access is more or less random, more paging action could ensue because fewer pages can be kept in memory and more data is transferred per page fault. So this change is as likely to decrease utilization as it is to increase it.

10.10 Consider the two-dimensional array A:

```
int A[][] = new int[100][100];
```

where A[0][0] is stored at location 200, in a paged memory system with pages of size 200. A small process resides in page 0 (locations 0 to 199) for manipulating the A matrix; thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement, and assuming page frame 1 has the process in it, and the other two are initially empty:

a. `for(int j = 0 ; j < 100 ; j++)`

```
    for( int i = 0 ; i < 100 ; i++ )
```

```
        A[i][j] = 0;
```

b. `for(int i = 0 ; i < 100 ; i++)`

```
    for( int j = 0 ; j < 100 ; j++ )
```

```
        A[i][j] = 0;
```

Solution:

a. In this case, the array A is accessed row by row and thus each row generates 2 page faults as the first reference to a page always generates a page fault. Using LRU, it will generate 200 page faults.

b. In this case, the array A is accessed column by column and thus the process references 100 pages in each outside loop (I), which is the working set of the program. But we only have 2 frames, and thus each array reference will generate a page fault. Using LRU, it will generate $100 * 100 = 10,000$ page faults.

This example shows that a well-written program can be much faster than a program is not carefully written.

10.11 Consider the following page-reference string:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames? Remember that all frames are initially empty, so your first unique pages will all cost one fault each.

* LRU replacement.

* FIFO replacement.

* Optimal replacement.

Solution:

Number of frames	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

10.12 Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or work-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware. Calculate the cost of doing so.

Solution:

You can use the valid/invalid bit supported in hardware to simulate the reference bit. Initially set the bit to invalid. On first reference a trap to the operating system is generated. The operating system will set a software bit to 1 and reset the valid/invalid bit to valid.

If this question interests you at all, there is an excellent paper by Bill Joy on this topic:

"Converting a Swap-based System to do Paging in an Architecture Lacking Page-Referenced Bits."

10.14 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

Solution:

In general this would not be an especially effective policy. By using this algorithm, you would gain many more free pages in memory that can be used by newly faulted pages. However, this comes at the expense of having to bring evicted pages back into memory that LRU or second-chance would not have evicted in the first place. This would be effective in workloads where a page is used once and then never again, because it wouldn't matter if the page was discarded.

10.16 A page-replacement algorithm should minimize the number of page faults. We can do this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages that are associated with that frame. Then, to replace a page, we search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address the problems of (1) what the initial value of the counters is, (2) when counters are increased, (3) when counters are decreased, and (4) how the page to be replaced is selected.

b. How many page faults occur for your algorithm for the following reference string, for four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

Solution:

a. Define a page-replacement algorithm addressing the problems of:

i. Initial value of the counters: 0.

ii. Counters are increased whenever a new page is associated with that frame.

iii. Counters are decreased whenever one of the pages associated with that frame is no longer required.

iv. How the page to be replaced is selected: find a frame with the smallest counter. Use FIFO for breaking ties.

b. 14 page faults

c. 11 page faults

10.17 Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of the remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

Solution:

effective access time = $(0.8) \cdot (1 \text{ microsec}) + (0.18) \cdot (2 \text{ microsec}) + (0.02) \cdot (20002 \text{ microsec})$

$$= 401.2 \text{ msec}$$

$$= 0.4012 \text{ millisec}$$

10.18 Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of CPU and the paging disk. The results are one of the following alternatives. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

- a. CPU utilization 13 percent; disk utilization 97 percent.
- b. CPU utilization 87 percent; disk utilization 3 percent.
- c. CPU utilization 13 percent; disk utilization 3 percent.

Solution:

- a. Thrashing is occurring.
- b. CPU utilization is sufficiently high to leave things alone, the system appears to run CPU-intensive jobs.
- c. Increase the degree of multiprogramming.

10.20 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

Solution:

Thrashing is caused by underallocation of the minimum number of pages required by a process, forcing it to continuously page fault. When a system thrashes, processes spend almost the entire time paging in and out from the disk, rather than doing useful computation. 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 (e.g. by arbitrarily picking some jobs to swap out temporarily until some of their competitors finish.).

8

Problem 1:

Consider a very simple file system for a tiny disk. Each sector on the disk holds 2 integers, and all data blocks, indirect blocks, and inodes are 1 disk sector in size (each contains 2 integers). All files stored on disk are interpreted as directories by the file

system (there are no "data files"). The file system defines the layout for the following data types on disk:

inode = 1 pointer to a data block + 1 pointer to indirect block

indirect block = 2 pointers to data blocks

directory = a regular file containing zero or more pairs of integers; the first integer of each pair is a file name and the second is the file's inumber

The value "99" signifies a null pointer when referring to a disk block address or directory name. An empty directory has one disk block with the contents "99 99". The inumber for root directory is "/" is 0.

The following data are stored on disk:

inode array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	7		8			3									
6	99		99			99									

disk blocks:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	32		96			1	99	99		57					
	3		1			99	99	99		6					

How many entries can appear in a maximum-sized directory? (Each entry is a pair of integers)

List all directories stored on this disk (full path names) along with the names of the files stored in each directory.

directory path name	inumber	indirect blocks	data blocks	contents (subdirectories)
/	0			

--	--	--	--	--

Modify the above data structures to add an empty directory called "87" to directory "/"

directory path name	inumber	indirect blocks	data blocks	contents (subdirectories)
/87				

Solution:

a. 3 (one data block and two other data blocks pointed by the one indirect block)

b.

directory path name	inumber	indirect blocks	data blocks	contents (subdirectories)
/	0	6	10 1	/32, /57
/32	3	n/a	8	n/a
/57	6	n/a	3	/57/96
/57/96	1	n/a	7	n/a

c.

directory path name	inumber	indirect blocks	data blocks	contents (subdirectories)
/87	9	n/a	14	n/a
/	0	6	10 1 13	/32, /57, /87

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

10	7		8			3			14						
6	99		99			99			99						

disk blocks:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	32		96			1	99	99		57			87	99	
	3		1			13	99	99		6			9	99	

Problem 2:

The MegaGiga hard disk rotates at 5400 rpm with an arm seek time given by $= 4 + 0.05t$ msec, where t is the number of tracks the arm seeks. Assume a block size of 512 bytes, and 1024 tracks with 512 sectors/track. The disk controller and DMA read or write data from/to disk at a rate of 4MB/sec.

What is the storage capacity of the disk?

Assume that we are reading a 16K-bytes file where the sectors happen to be allocated contiguously on the same track. Compute the maximum and minimum disk throughput that is possible while reading the file?

Solution:

1)

The storage capacity of the disk is

(sector size) * (number of sectors/track) * (number of tracks)

- Assume that we are reading a 16K-bytes file where the sectors happen to be allocated contiguously on the same track. Compute the maximum and minimum disk throughput that is possible while reading the file?

2)

Maximum disk throughput occurs when the head is right on top of the required sector. In this case we can transfer the 32 sectors that constitute the file immediately without seeking or paying rotational latency overhead. But we do not write instantaneously, we need to compute the time it takes for the head to travel over the 32 sectors. This time can be computed as follows:

It takes the rotational latency for the head to travel 512 sectors (sectors per track). Therefore from one sector to the next = rotational latency/512

$$= (60/5400) / 512$$

$$= 22 \text{ microseconds (rounded up)}$$

Therefore, to get the 16K bytes out of the disk, it takes $32 \text{ sectors} * 22 \text{ microseconds/sector} = 704 \text{ microseconds}$. Note that the bandwidth of the disk itself is therefore about 23.3MB/s. But, we also have to pay the DMA overhead, which is

$$16K / (4 * 1024 * 1024) = 3.9 \text{ msec}$$

So, the total time is $3.9 \text{ ms} + .704 \text{ ms} = 4.604 \text{ ms}$. The transfer time is dominated by the DMA overhead, and the throughput is slightly less than the 4MB/sec rate of the DMA controller ($4.604 \text{ ms per } 16KB = 3.56 \text{ MB/s}$).

Moral of the story: When we make a transfer and the disk head does not have to move, then we transfer data at the maximum speed that the disk controller electronics allow.

Minimum throughput: This will occur when the disk head has to travel the maximum distance to get to the data. In this case, at worst, it has to go over the entire platter, and then wait for an entire rotation before it gets to the beginning of the file. The time to do this is:

$$\text{seek time} + \text{latency time} = (4 + 0.05 * 1024) + (60/5400)$$

$$= 55.2 \text{ (seek)} + 11.11 \text{ (rotation)}$$

$$= 66 \text{ msec}$$

The DMA overhead is the same as above. Therefore, the overhead is dominated by the cost of seeking and waiting for the head to get to the right sector on the track. The result is $16 \text{ KB}/66\text{ms} = 247\text{KB/sec}$ (about 7% of the sequential throughput).

Moral of the story: When the head has to move, the throughput drops considerably, and you can see that the effect may produce only 7% of the controller bandwidth on a read or write (and with a better DMA controller the peak bandwidth would be higher and the long seek and rotation would reduce the bandwidth to about 1% of peak).

Exercise: Compute the same if the sectors were not contiguous. You will see that 7% of the max bandwidth may not be even reachable if the sectors are scattered throughout the disk. Sequential allocation is good!!

Problem 3:

The FastFile file system uses an inode array to organize the files on disk. Each inode consists of a user id (2 bytes), three time stamps (4 bytes each), protection bits (2 bytes), a reference count (2 byte), a file type (2 bytes) and the size (4 bytes). Additionally, the inode contains 13 direct indexes, 1 index to a 1st-level index table, 1 index to a 2nd-level index table, and 1 index to a 3rd level index table. The file system also stores the first 436 bytes of each file in the inode.

Assume a disk sector is 512 bytes, and assume that any auxiliary index table takes up an entire sector, what is the maximum size for a file in this system.

Is there any benefit for including the first 436 bytes of the file in the inode?

Solution:

Note: The first thing you need to determine is the size of an index (2 bytes or 4 bytes). Given that we have a 3-level indexing scheme, you can quickly compute the number of sectors that you can get by 2 byte index and 4 bytes indexes. You will see that the 2-byte indexing does not work. This would give you up to 256 indexes per sector, with a maximum file size of $436 + 13 * 512 + 1 * 256 * 512 + 1 * 256 * 256 * 512 + 1 * 256 * 256 * 256 * 512 = \text{whatever}$. The problem with this analysis is that you have far more disk sector for this scheme to work than can be encoded in 2 bytes. You can use 3 bytes, but this can get ugly. So, we go with 4-byte indexes, giving us 128 indexes per sector, and the correct answer is: $436 + 13 * 512 + 1 * 128 * 512 + 1 * 128 * 128 * 512 + 1 * 128 * 128 * 128 * 512 = 1082203060$, roughly 1G.

Yes. Most files are small. If the file size is 436 bytes or less, then the entire file can be read and written in one disk operation without having to do a separate access to the inode.

Theory Question 1:

Is it fundamentally necessary to store on disk information about the unallocated disk sectors? Explain why.

Solution:

No. This information can be computed at startup by traversing the file system tree. However, it is a safety issue. The job of a utility like fsck is simplified and is rendered more secure by having such a list.

Theory Question 2

Pooh Software Ltd. is selling a file system that uses a UNIX-like scheme with multi-level indexing. For more reliability, the inode array is actually replicated on the disk in two different places. The intent is that if one or a group of sectors that are storing either replica of the array become bad, the system can always recover from the replica. Discuss the effect of having this replicated data structure on performance.

Solution:

Updates to inodes will have to be done to both copies. This will decrease the performance of operations that attempt to modify the inodes, such as allocating a new file, deleting a file, extending the size of a file, or opening a file for updates (among perhaps many others). However, reading information from an inode can be made faster with clever disk scheduling. To read a particular inode, we schedule a read from either track that is closest to the current head position.

Theory Question 3

Contiguous allocation of files leads to disk fragmentation. Is this internal or external fragmentation?

Solution:

Both! A file is still allocated in disk block units, which means the last block in a file would be half empty (internal fragmentation). Additionally, contiguous allocation results in external fragmentation because it could be the case that there many small unallocated fragments between files and they cannot be used to allocate a file whose size is greater than the size of any fragment, but less than the total size of the free space.

Theory Question 4

Consider an indexed file allocation using index nodes (inodes). An inode contains among other things, 7 indexes, one indirect index, one double index, and one triple index.

What usually is stored in the inode in addition to the indexes?

What is the disadvantage of storing the file name in the inode? Where should the file name be stored?

If the disk sector is 512 bytes, what is the maximum file size in this allocation scheme?

Suppose we would like to enhance the file system by supporting versioning. That is, when a file is updated, the system creates new version leaving the previous one intact. How would you modify the inode allocation scheme to support versioning? Your answer should consider how a new version is created and deleted.

In a file system supporting versioning, would you put information about the version number in the inode, or in the directory tree? Justify your answer.

Solution:

An inode usually stores indexes and:

the file size in bytes,

special flags to indicate if the file is of a special kind (directory, symbolic links)

time stamps (creation date, modification date, last read date),

a reference count of how many names refer to that file from the name space,

owner identification (e.g. in UNIX, user id and group id), and

security credential (who should be able to read the file)

Storing the file name in the inode limits the flexibility of the file system and precludes the use of hard links. Also, since it is desirable to have relatively long file names, it would be cumbersome to store variable size character arrays in the inode structure (or wasteful if a certain maximum is defined).

We must first determine the size of an index. For a 2-byte index, we can have 65536 disk blocks, i.e. $512 * 65536 = 32\text{MB}$. But a triple index structure can express more disk blocks. Therefore, we go to a 4-byte indexing scheme (3-byte indexing scheme are not attractive and are not sufficient). A 4-byte indexing scheme therefore gives a maximum file size of $7*512 + 128*512 + 128*128*512 + 128*128*128*512 = 108219952$ or about 1Gbytes

We augment the inode structure such that different versions of the file share the common blocks. When a new version is created, we copy the information from the older version into the new one. Also, we will need to add a reference count to every disk block, and set that reference count to 1 when a block is allocated to a single file. The reference count is incremented whenever a block is shared among different versions. Then, when a version is modified, we perform a copy-on-write like policy and decrement the reference count of the disk blocks and actually copy the modified disk blocks and make them only accessible to the new version.

It is better put in the directory space in order to make it easier for the user to relate the different versions of the same file.

Theory Question 5

Versioning: It is often desirable for users to maintain different versions of the same file (for example, during program development, for recovering in the case of wrong updates applied to a version, etc.). The VMS file system implements versioning by creating a new copy for the file every time it is opened for writing. The system also supported versioning in the naming structure, appending a version number to the name of each file. For instance, when `foo.c;1` is updated, the system creates `foo.c;2`, etc. The system implemented commands to manipulate versions, for example, the `PURGE` command would delete all versions except the most recent one. Also, users could disable versioning for some files. Discuss the pros and cons for this scheme.

Solution:

The main advantage of this scheme is its simplicity. There really is no support that is fundamentally needed from the operating system. Instead, application programs and the shell can be all linked with a modified library that creates the new version whenever a file is open for update. The creation of the file can be done with the available system calls. For example, in a UNIX-like system this can be implemented by having the `open()` library stub actually calls `creat()` with the new name, then do the copy, then open the newly created file.

The obvious disadvantage of this scheme is performance and overhead. Opening a file to update a single byte would have to make an entire file copy. Also, there is an overhead in storage in that even though multiple versions of the same file have common information, they may not attempt to share the disk blocks that contain the common information. This results in a waste in space. However, such sharing is possible only with non-trivial modifications to the file system.

9

12.1 Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.

- a. The block is added at the beginning.
- b. The block is added in the middle.
- c. The block is added at the end.
- d. The block is removed from the beginning.
- e. The block is removed from the middle.
- f. The block is removed from the end.

Solution:

	Contiguous	Linked	Indexed
a.	201	1	1
b.	101	52	1
c.	1	3	1
d.	198	1	0
e.	98	52	0
f.	0	100	0

12.3 What problems could occur if a system al

lowed a file system to be mounted simultaneously at more than one location?

Solution:

There would be multiple paths to the same file, which could confuse users or encourage mistakes (deleting a file with one path deletes the file in all the other paths).

12.5 Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

Solution:

Contiguous: if file is usually accessed sequentially, if file is relatively small.

Linked: if file is large and usually accessed sequentially.

Indexed: if file is large and usually accessed randomly.

12.6 Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

Solution:

Let Z be the starting file address (block number).

a. Contiguous. Divide the logical address by 512 with X and Y the resulting quotient and remainder respectively.

- i. Add X to Z to obtain the physical block number. Y is the displacement into that block.
- ii. 1

b. Linked. Divide the logical physical address by 511 with X and Y the resulting quotient and remainder respectively.

- i. Chase down the linked list (getting $X + 1$ blocks). $Y + 1$ is the displacement into the last physical block.
- ii. 4

c. Indexed. Divide the logical address by 512 with X and Y the resulting quotient and remainder respectively.

- i. Get the index block into memory. Physical block address is contained in the index

block at location X. Y is the displacement into the desired physical block.

ii. 2

<BR

12.7 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area (of a specified size). If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

Solution:

This method requires more overhead than the standard contiguous allocation. It requires less overhead than the standard linked allocation.

12.9 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

Solution:

Caches allow components of differing speeds to communicate more efficiently by storing data from the slower device, temporarily, in a faster device (the cache). Caches are, almost by definition, more expensive than the device they are caching for, so increasing the number or size of caches would increase system cost.

SCHEDULING:

S3(e). Run P1 for 8 time units, run P3 for 1 time unit, run P2 for 5 time units, run P5 for 2 time units, and run P4 for 3 time units.

(f). same answer as S3(3). In preemptive priority, a process, say process P3, which is at the same priority as a running process, say P0, never preempts the running process. If the priorities of processes are fixed, the only time preemption occurs is if a newly ready process has a higher priority than the currently running process.

S5.

The order of execution is

Time Interval	0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10	10-11	11-12
Gantt	P1	P1	P2	P2	P3	P3	P4 ///	P5	P5	P1 ///	P6 ///	P2
Q0	P2	P2	P3	P3, P4	P4	P4	P5					
Q1			P1	P1	P1, P2	P1, P2	P1, P2, P3	P1, P2, P3	P1, P2, P3	P2, P3, P5	P2, P3, P5	P3, P5
Q2												

12-13	13-14	14-15	15-16	16-17	17-18	18-19	19-20	20-21	21-22	22-23	23-24	24-25	25-26
P2	P3 ///	P5	P5	P2	P2	P2	P7	P7	P7	P7 ///	P5 ///	P2	P2 ///
					P7	P7							
P3, P5	P5												
	P2	P2	P2	P5	P5	P5	P5, P2	P5, P2	P5, P2	P5, P2	P2		

S6. (a)

i) not done here

ii) Turnaround time = completion time - arrival time (This is the same as T in examples given in lectures.)

A) FCFS

$$P1 = 3 - 0 = 3$$

$$P2 = 5 - 0 = 5$$

$$P3 = 14 - 0 = 14$$

$$P4 = 15 - 0 = 15$$

$$P5 = 19 - 0 = 19$$

B) and C) SJF and SRT act the same on this problem

$$P1 = 6 - 0 = 6$$

$$P2 = 3 - 0 = 3$$

$$P3 = 19 - 0 = 19$$

$$P4 = 1 - 0 = 1$$

$$P5 = 10 - 0 = 10$$

D) and E) Preemptive and nonpreemptive priority act the same on this problem

$$P1 = 15 - 0 = 15$$

$$P2 = 2 - 0 = 2$$

$$P3 = 12 - 0 = 12$$

$$P4 = 3 - 0 = 3$$

$$P5 = 19 - 0 = 19$$

F) RoundRobin

$$P1 = 10 - 0 = 10$$

$$P2 = 7 - 0 = 7$$

$$P3 = 19 - 0 = 19$$

$$P4 = 4 - 0 = 4$$

$$P5 = 14 - 0 = 14$$

c) Waiting time = Turnaround time - Burst time ($= T - t = M$) (This is the same as Missed time.)

A) $P1 = 3 - 3 = 0$

$$P2 = 5 - 2 = 3$$

$$P3 = 14 - 9 = 5$$

$$P4 = 15 - 1 = 14$$

$$P5 = 19 - 4 = 15$$

B) $P1 = 6 - 3 = 3$

$$P2 = 3 - 2 = 1$$

$$P3 = 19 - 9 = 10$$

$$P4 = 1 - 1 = 0$$

$$P5 = 10 - 4 = 6$$

C) $P1 = 15 - 3 = 12$

$$P2 = 2 - 2 = 0$$

$$P3 = 12 - 9 = 3$$

$$P4 = 3 - 1 = 2$$

$$P5 = 19 - 4 = 15$$

$$D) P1 = 10 - 3 = 7$$

$$P2 = 7 - 2 = 5$$

$$P3 = 19 - 9 = 10$$

$$P4 = 4 - 1 = 3$$

$$P5 = 14 - 4 = 10$$

S6 (b) Average Waiting Time = Total Waiting time / Number of Processes = $\text{sum}(M)/5$

$$\text{FCFS: } (0 + 3 + 5 + 14 + 15) / 5 = 7.4$$

$$\text{SJF/SRT: } (3 + 1 + 10 + 0 + 6) / 5 = 4$$

$$\text{Priority: } (12 + 0 + 3 + 2 + 15) / 5 = 6.4$$

$$\text{RR: } (7 + 5 + 10 + 3 + 10) / 5 = 7$$

The minimal waiting time occurs with the shortest job first and the shortest remaining time algorithms.

S6 (c)

When all processes start at time 0, shortest job first behaves identically to shortest remaining time because if we execute for some time on a process, we will never change our mind about which is the shortest process. Similarly, preemptive and nonpreemptive priority will behave the same in this case because once we have picked a higher priority process, based on knowledge of all processes, we will never have any reason to change our mind and preempt it.

By the way, the priority values are NOT used in the feedback algorithm. Priority is informally indicated by which queue a process is in, but no explicit priority values are used. Also, with the feedback algorithm, if a process does not use its full quantum, when the next process is scheduled, give it the full quantum.

S7.

The order of execution is

Time Interval	0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10	10-11	11-12
Process	P1	P2	P3	P4 ///	P1 //	P2	P2	P5	P3	P3 ///	P6 ///	P5
Q0	P2						P5					
Q1		P1	P1, P2	P1, P2, P3	P2, P3	P3	P3	P3	P5	P5	P5	
Q2								P2	P2	P2	P2	P2

12-13	13-14	14-15	15-16	16-17	17-18	18-19	19-20	20-21	21-22	22-23	23-24	24-25
P5	P2	P2	P2	P5	P5 ///	P7	P7	P7	P2	P2	P2	P7
					P7							
P2	P5	P5	P5	P2	P2	P2	P2	P2	P7	P7	P7	

S8.

Disabling interrupts frequently could affect the system's clock if the system's clock depended on the timer interrupt. In some cases, time is determined by counting the number of timer interrupts. If "interrupts were disabled," then all interrupts would be

disabled, including the timer interrupt. Thus, the time would not be updated on the proper, regular basis. The effects could be reduced if the maximum time that interrupts could be disabled was less than the time between successive timer interrupts (at most one tick would be late and no ticks would be lost) or if special (not usual) hardware was used that allowed multiple interrupts of the same type to be kept pending.

S9.

The question says that when process P356, which frequently disables interrupts, is run, interactive users complain about the computer being slow. This relationship exists because the timer interrupt is used to keep track of the quantum for the multilevel feedback queue algorithm. If "interrupts are disabled frequently," then all interrupts are frequently disabled, including the timer interrupt. Typically, the running process is given a quantum of ticks when placed on the processor. Each time, a timer interrupt occurs, the number of ticks remaining is reduced by one. When the number of ticks reaches zero, the quantum is over and the scheduler is executed. If the timer interrupts are disabled, then the running process can run as long as it wants without being preempted. Ordinarily, only one pending timer interrupt is held by the hardware, so the ticks are completely lost. Process P356 will get longer CPU bursts than other processes and it will move down the multilevel queues more slowly as a direct result. Interactive processes depend on getting service from the CPU with little delay. Whenever process 356 has interrupts turned off, no interactive process will get service until it is turned back on.

CONCURRENT PROCESSES AND DEADLOCK:

C1. (a) Suppose we have two processes, called P1 and P2, and P1 runs first. You can think of i and j as constants that are not shared between the processes. In P1, $i = 1$ and $j = 2$; in P2, $i = 2$ and $j = 1$.

There are three shared variables, $\text{flag}[1]$, $\text{flag}[2]$, and turn . Before the processes start, $\text{flag}[1] = \text{false}$, $\text{flag}[2] = \text{false}$, and $\text{turn} = 1$.

When execution begins, suppose process P1 runs first.

Process P1 executes the line:

```
turn = j
```

which sets turn to 2.

Then suppose P1 is preempted. (Preemption can happen after any line, but for this exercise, this is the place that causes the algorithm to fail to solve the critical section problem.)

Process P2 runs and executes several lines:

```
turn = j      // which sets turn to 1
```

```
flag[i] = true // which sets flag[2] = true
```

```
while (flag[j] and turn = j)
```

```
// which checks whether flag[1] is true
```

```
// and turn is 1
```

But flag[j] isn't true,

i.e., flag[1] was originally false and we haven't changed it.

So, process P2 does not wait in the while loop doing no-op.

Instead process P2 enters the critical section

Now suppose process P2 is preempted.

When process 1 resumes execution, it executes the lines:

```
flag[i] = true // which sets flag[1] to true
```

```
while (flag[j] and turn = j)
```

// which checks whether flag[2] is true and

// turn is 2

while and finds that it is false

But turn isn't 2 because process P2 set turn to 1.

So, process P1 does not wait in the while loop doing no-op.

Instead process P1 enters the critical section.

Now there are two processes that are both in the critical section. The algorithm has failed to solve the critical section problem because it allowed more than one process to be executing in the critical section at the same time.

(b) not done here.

C6.

If you disallow interrupts, no other instructions can be run besides those that are modifying a shared variable. As well, disabling interrupts decreases system efficiency with increasing message passing. The clock will lose time when an interrupt is pending and then a new interrupt arrives so the pending one is lost.

To minimize the effects, software should be designed so that the instructions run only as long as the time between the timer interrupts. Otherwise, a pending interrupt may be lost when a new one arrives.

C7.

The checker process:

wait(wrtcnt);

signal(wrtcnt);

wait(mutex);

rdcnt := rdcnt + 1;

if rdcnt = 1 then wait(wrt);

signal(mutex);

...

reading is performed

...

wait(mutex);

rdcnt := rdcnt - 1;

if rdcnt = 0 then signal(wrt);

signal(mutex);

The updater process:

wait(wrtcnt);

wait(wrt);

...

writing is performed

...

signal(wrt);

signal(wrtcnt);

C7 a) not done here

b) Yes, a deadlock could exist. “If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.”

All four necessary conditions are true:

- i) mutual exclusion is always assumed
- ii) hold and wait: P2 holds R2 but also requests R1
- iii) no preemption: the question assumed this
- iv) P2 wants R3 but R3 is held by P5, P5 wants R2 but R2 is held by P2, P2 wants R1 but R1 is held by P2.

(Silberschatz & Galvin, OS Concepts, 6th edition, pp. 250-253; 5th edition, pp. 220-221)

C9 a) not done here

b) Yes, a deadlock could exist. “If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.”

All four necessary conditions are true:

- i) mutual exclusion is always assumed
- ii) hold and wait: P2 holds R4 but also requests R2
- iii) no preemption: the question assumed this

iv) P2 wants R2 but R2 is held by P3, P3 wants R4 but R4 is held by P2.

C10

P1 requests R1, succeeds and P1 is given R1

P3 requests R3, succeeds and P3 is given R3

P1 requests R2, is refused because P1 is holding R1, which is a higher priority resource.

P1 frees R2, is refused because it is not holding R2

P2 requests R2, succeeds and P2 is given R2

P2 frees R2, succeeds.

P3 requests R2, is refused because it is holding R3, which has a higher priority.

FILE MANAGEMENT:

F1. Having an operating system pay attention to file types is a good feature because the OS can avoid certain errors. For example, the OS can prevent the printing of non-ASCII “garbage characters,” which are present in the binary-object form of a program. As well, if the file has an extension, this offers a “hint” as to what applications can operate on specific files. However, the extension may not always be required. On the other hand, the size of your OS may have to be very large in order to execute various applications on different file types. Netscape avoids this by calling application programs (according to file type) found outside of Netscape code. Another disadvantage may be that since every file created must be defined as one of the file types currently supported by the OS; adding new applications and file types may require cost or effort to add to the OS, but unless

they are added, the files of the new file type cannot be used. (Sec. 11.1.3 File Types, OS Concepts, Silberschatz, Galvin, and Gagne)

F2. The relationship between $A(X, Y)$ and $A(Z, Y)$ for an arbitrary object y is that $A(X, Y)$ is a subset of $A(Z, Y)$.

(The mathematical symbol is a sideways U opening to the right, with a line under it.)

X possesses at most the same abilities as its parent.

Therefore, the privileges of X are a subset of those of Z.

(Note this is NOT a proper subset, which is indicated by a sideways U opening to the right.)

F3. Advantages of implicitly opening/closing a file:

simpler programming because no open and close statements are required

the user can't forget to close a file.

file sharing / file locking can be completely controlled by the OS which may simplify programming

Advantages of explicitly opening/closing a file:

access to a file can be freed as soon as a program is done with it, without waiting for the program to terminate. Two benefits:

- user can control the time when the file on disk is completely updated. For example, if the same program runs for many days, its output may not be transferred to disk until the file is explicitly closed. If this happens, the file may be overwritten by another user before it is finally closed (typically the other user's changes are then lost). Also, other users may obtain out-of-date information because the file has not been properly closed.
- OS can reuse resources such as "open file" table entries

complex file sharing protocols can be implemented by the users

may give them the behavior they want

F4.

a) To provide access to 4990 out of 5000 users in UNIX, a group should be created which lists the 4990 names. If a person's name is not on the list, they cannot access the file. The group in the access list must be associated with the file. Only the manager of the facility can make changes to groups in UNIX.

b) A more convenient protection scheme would allow the inverse. That is, we would create a list of all the users who are not allowed access to this file and associate this list with the file.

F5.

(a) Diagrams not shown. Assume you can identify the correct list in constant $O(1)$ time. Then access lists require on average $10/2 = 5$ items to be accessed, because we identify the correct list for the file in $O(1)$ time and then examine the access list, which contains 10 items. Capability lists require on average $1000/2 = 500$ items to be accessed, so access lists are faster.

(b) Diagrams not shown. Access lists require on average $1000/2 = 500$ items to be accessed, because we identify the correct list for the file in $O(1)$ time and then examine the access list, which contains 1000 items. Capability lists require on average $10/2 = 5$ items to be accessed, so capability lists are faster.

F6.

Contiguous (the size field for the file will be adjusted in main memory for all cases):

a) Because there is no room to insert before the starting block, you must move all 600 blocks down and write 1 new block.

$$600(1r + 1w) + 1w = 600r + 601w = 1201 \text{ I/O operations}$$

b) Insert after 200 and before 201. Therefore, move blocks 201 to 600 down and write the new block.

$$400(1r + 1w) + 1w = 400r + 401w = 801 \text{ I/O operations}$$

- c) Go to end and write one new block.

$$1w = 1 \text{ I/O operation}$$

- e) Go to block 200 and move up remaining 400 blocks.

$$400(1r + 1w) = 800 \text{ I/O operations}$$

- f) No action required to delete the last block other than updating the file size as usual.

$$0 \text{ I/O operations}$$

Linked:

- a) Set the link of the new block (in main memory) to point to the current starting block number, write this one new block to disk, set the starting block number to the new block's number.

$$1w = 1 \text{ I/O operation}$$

- b) We cannot find block 200 without traversing the linked list stored in the first 199 blocks. So, we first read through these 199 blocks. Next, read block 200, copy its link into new block, write block 200, and write the new block.

$$199r + 1r + 1w + 1w = 200r + 2w = 202 \text{ I/O operations}$$

- c) Traverse (read) first 599 blocks, read block 600, update its link, write block 600, and write new block.

$$600r + 1w + 1w = 602 \text{ I/O operations}$$

- e) Read first 199 blocks, read block 200, update links for block 199, and write block 199.

$$199r + 1r + 1w = 200r + 1w = 201 \text{ I/O operations}$$

- f) Read through first 599 blocks and then update link in block 599.

$$599r + 1w = 600 \text{ I/O operations}$$

Indexed:

- a) Update index in main memory and write new block.

1w = 1 I/O operation

b) Update index in main memory and write the new block.

1w = 1 I/O operation

c) Update index in main memory and write one new block.

1w = 1 I/O operation

e) Remove block address from linked list in index block. The list is in main memory so there is no need for read and write operations required.

0 I/O operations

f) Simply remove the block's address from the linked list in main memory in the index block.

0 I/O operations

F6'. Answer not provided.

F7. Assumptions:

- Assume an index is small enough to fit into a single block. (In fact, a 512 block file will probably require more than a single 512 byte block because block addresses typically require 3-4 bytes each.).
- For part (a), a logical address is specified as a byte offset within a file and a physical address is specified with a physical block number and an offset in that block.

Suppose we assume that the blocks in the linked list are numbered from 0. In other words, the first block in the linked list is logical block 0, the second is logical block 1, etc.

We can determine the logical block number LB by dividing the logical address L by the logical block size. If logical block size is 512, then

$$LB := \text{trunc}(L / 512)$$

Linked:

a) $512 \text{ bytes} - 4 \text{ bytes (for a pointer)} = 508 \text{ bytes/block}$.

[This is inconsistent with the question, but for a linked scheme it is impossible to have the same logical and physical sizes for a block.]

We determine the logical block number LB by dividing the logical address L by the logical block size.

$$LB := \text{trunc}(L / 508)$$

If $LB = 0$, then the starting block number indicates the correct block. Otherwise, you must traverse the linked list from the beginning and keep count of how many blocks have been travelled through. When you are on logical block $LB - 1$ (you are on the LBth block in the list), the link of this block indicates the correct logical block.

Within the correct block, the logical offset (LO) is:

$$LO := L \bmod 508.$$

b) Logical block number 4 means to start counting at the beginning of the linked list and read each block until you have read the 5th block in the list (logical block 4). Thus, you must read 5 physical blocks.

Alternate Answer for Linked:

(a) If instead we assume that the logical blocks are numbered from 1, the answer is:

We determine the logical block number LB by dividing the logical address L by the logical block size and adding 1.

$$LB := \text{trunc}(L / 508) + 1$$

If $LB = 1$, then the starting block number indicates the correct block. Otherwise, you must traverse the linked list from the beginning and keep count of how many blocks have been travelled through. When you have counted to LB, you are on the correct block; or when you have counted to $LB - 1$, the link of this block indicates the correct logical block.

Within the correct block, the logical offset (LO) is:

$$LO := L \bmod 508.$$

b) Logical block number 4 means to start counting at the beginning of the linked list and read each block until you have read the 4th block in the list (logical block 4). Thus, you must read 4 physical blocks.

Indexed:

Assuming the array positions are numbered from 0

a) array position in index $:= \text{trunc}(L / 512)$

Look up the value in this position in the index and go to the block number found in this location. You have direct access (i.e., do not need to traverse any lists).

The offset in this block is

$$\text{offset} := L \bmod 512$$

b) Logical block 4 means look at position 4 in the index array in main memory. You go directly to the address listed in array[4]

(i.e., you do not have to traverse the list). Therefore, you read only one physical block.

F8. Answer not provided.

F9. (a) We note that a data block can be used as an index block. It can hold the block pointers for $4096/4 = 1024$ blocks, since it has 4096 bytes and each block pointer takes 4 bytes. Maximum file size uses all possible pointers.

$$\text{number of blocks} = 10 + (4096/4) + (4096/4)^2 + (4096/4)^3$$

$$= 10 + 1K + 1M + 1G$$

maximum size = number of blocks * block size, where block size is 4Kb

$$= 40Kb + 4Mb + 4Gb + 4Tb$$

(b)

$$\text{number of blocks} = 10 + 1K + 1M + 1G + 1T, \text{ and block size} = 4Kb$$

maximum size = 40Kb + 4Mb + 4Gb + 4Tb + 4Pb

(c) a block pointer gives the id (number) of a block, which is an integer. From part (b), we know the number of blocks in the largest file is

$$40 + 1K + 1M + 1G + 1T$$

$$= 40 + 2^{10} + 2^{20} + 2^{30} + 2^{40}$$

However, in 4 bytes, there are only $4 \times 8 = 32$ bits, so at most 2^{32} different numbers (block ids) can be represented in 4 bytes. Thus, a 4-byte wide block pointer is not adequate to address all blocks in the largest file described in part (b).

F10. (a) not shown.

(b) Words of the bit vector are checked sequentially. The first word contains 0 bits for blocks 0 through 31, the second word contains 0 bits for blocks 32 through 63, and the third word has a 1 in it, in the fourth bit for block 67. Thus, $\text{ceiling}(67+1/32) = \text{ceiling}(2.x) = 3$ words must be checked.

(c) not shown. The four blocks may appear in any order.

(d) Only one block must be checked because the free list has only free blocks on it. Since any free block will do, we take the first one regardless of the ordering.

SOLUTIONS TO SELECTED EXERCISES:

INTRODUCTION:

1.

(a) operating system:

(b) batch operating system: an operating system that executes a stream of jobs, where each job provides control instructions for itself, with no provision for interaction between the user and job while the job is executing.

(c) control card: a punched card with control information about the nature of the job that follows, e.g., control card might set maximum run time.

(d) turnaround time:

(e) spooling: spooling means simultaneous peripheral operation on-line.

Spooling uses the disk as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

(f) job pool: in early OSES, this was a data structure that described the set of jobs that have been read (from cards or tape) onto disk and are ready to run.

(g) multiprogramming: is running the computer system with multiple processes (i.e., executable programs or jobs) loaded in memory at the same time, and having the CPU switch execution among them to try to keep the CPU utilized as fully as possible.

(h) time sharing: allow many users to share the computer simultaneously by using CPU scheduling and multiprogramming, with the switching between processes happening so fast that the users can all interact with the computer in real time.

(i) MULTICS: early OS (computing utility) developed by MIT and others in 1967-70 and used for many years thereafter, which influenced the development of UNIX.

(j) multiprocessing: using computers with more than one CPU

(k) tightly coupled system: a multiprocessor machine in which the processors share the system clock, memory, and peripherals.

(l) loosely coupled system: a distributed system in which each processor has its own memory and communicates with other processors through external lines such as buses.

(m) symmetric multiprocessing

(k) asymmetric multiprocessing.

2.

(a) bootstrap program: a small program that runs every time the computer is started. It sets up the computer for the OS, loads the OS, and starts the OS executing.

(b) system call: a request to the operating system from a program; In the program, a system call looks like a call to a function in a programming language, e.g. C, but it actually causes a portion of the OS's code to be invoked.

(c) syscall instruction.

(d) interrupt: a signal from a hardware device to the CPU. The CPU finishes its current instruction, saves the current state of execution, executes the interrupt handling routine (OS code) for this type of interrupt, and then restores the state of execution.

(e) interrupt vector: an array/table that contains the addresses of the interrupt handling routines, indexed by the type of interrupt.

(f) interrupt service routine: also known as an interrupt handling routine: code that is executed to service (or handle) an interrupt of a given type. The address of this code is stored in the interrupt vector.

(g) trap: a trap or exception is an indication to the CPU that immediate action is requested due to a condition caused by software. Sources include errors, such as division by zero or attempting to access an unauthorized area of memory, and system calls.

(h) no answer provided.

(i) no answer provided.

(j) no answer provided.

(k) direct memory access (DMA): an efficient way of transferring data between a peripheral device and memory, where bytes of data are transferred directly to or from the device to memory. The CPU is involved only to start the transfer and to handle the interrupt generated when the transfer is done.

(l) volatile memory (volatile storage): any type of storage that loses its contents when the electrical power is turned off. Registers, cache, and main memory are all volatile.

(m) – (v) no answer provided.

3.

(a) change to monitor mode (system mode)

- must be protected because all privileges over the computer system are available in monitor mode, and the purpose of protection is to ensure that this power is not available at will to the user.

(b) change to user mode

- not in minimal set because a process never gains privileges by changing to user mode because this is the unprivileged state; the code written by the user should always be executing in this state already.

(c) turn on the timer/clock interrupt

- not in minimal set because turning on the interrupts ensures that the system is functioning normally, which it should always be when user code is executing.

(d) turn off the timer/clock interrupt

- must be protected because when interrupts are turned off, the scheduler is not invoked and the OS has no way of controlling the usage of time on the CPU.

(e) set the value of the timer/clock

- must be protected because the timer is used to determine the amount of time a process is allowed to stay on the CPU before being preempted; if the user code were allowed to change the timer, the timer could be repeatedly reset to allow a processor to obtain unlimited time on the CPU.

(f) read a value from monitor memory

- not in minimal set because OS's control of system is not jeopardized, but there are very strong security reasons for protecting it to prevent users from spying on other users

(g) read an instruction from monitor memory

- not in minimal set because OS's control of system is not jeopardized, but there are very strong security reasons for protecting this type of reading to prevent users from learning internal details of OS operation.

(h) write a value to monitor memory

- must be protected because otherwise user could change contents of OS, i.e., its instructions and data, and thereby obtain control of the system.

9.

Dual-mode operation forms the basis for I/O protection, memory protection and CPU protection. In dual-mode operation there are two separate modes, monitor mode (also called 'system mode' and 'kernel mode') and user mode. In monitor mode, the CPU can

use all instructions and access all areas of memory. In user mode, the CPU is restricted to unprivileged instructions and a specified area of memory. User code should always be executed in user mode and the OS design ensures that it is. When responding to system calls, other exceptions, and interrupts, OS code is run. This code can be run in monitor mode.

Input/output is protected by making all input/output instructions privileged. While running in user mode, the CPU cannot execute them; thus, user code, which runs in user mode, cannot execute them. User code requests I/O by making appropriate system calls. After checking the request, the OS code, which is running in monitor mode, can actually perform the I/O using the privileged instructions.

Memory is protected by partitioning the memory into pieces. While running in user mode, the CPU can only access some of these pieces. The boundaries for these pieces are controlled by base and limit registers (specifying bottom and top bounds). These registers can only be set via privileged instructions.

CPU usage is protected by using the timer device, the associated timer interrupts, and OS code called the scheduler. While running in user mode, the CPU cannot change the timer value or turn off the timer interrupt, because these require privileged operations. Before passing the CPU to a user process, the scheduler ensures that the timer is initialized and interrupts are enabled. When a timer interrupt occurs, the timer interrupt handler (OS code) can run the scheduler (more OS code), which decides whether or not to remove the current process from the CPU.

12. A system call is a request that can be made to the operating system from within a process (i.e., from within a running program). A system program is an executable program included in the “operating system” set of software and made available for the administrators or users to execute. A system program typically performs actions that make the system more convenient to use. Sleep is a system program that a user can start from within the UNIX shell:

```
% sleep 10
```

and `nanosleep` is a system call that might be called from within the process created when the sleep program is run.

Other examples of system programs:

```
% vi
```

```
% ls
```

```
% cp
```

% chmod

These system programs may call one or more system calls, e.g., 'ls' requires the 'stat' system call, while 'cp' uses the open/read/write/close system calls, and 'chmod' uses a system call that happens to have the same name, 'chmod'. See man 2 chmod

man 3 chmod

14. A virtual machine is a software emulation of a real or imaginary (hardware) machine. It is completely implemented in software. A virtual machine for the Intel 8086 processor is used to allow programs written for the 8086 to run on different hardware. The user has the advantage of not having to purchase or maintain the correct hardware if it can be emulated on another machine, such as a Sun. Java code is written for an imaginary machine called the Java Virtual Machine, which would be difficult to realize in hardware. The user has the advantage of not having to purchase special hardware because the Java virtual machine is available to run on very many existing hardware/OS platforms. As long as the Java Virtual Machines are implemented exactly according to specification, Java code is very portable since it can run on all platforms without change.

MEMORY MANAGEMENT:

M3.

a) answer not given here

b) None of the algorithms can allocate space for all four processes. However, in the above example, worst-fit makes the least efficient use of memory. This is because the algorithm uses the two largest spaces for two smaller processes. This space could be more efficiently used for a larger process. For this reason, worst-fit is unable to find space for two processes unlike best and first-fit, both of which are unable to allocate space for only one process.

M4.a) virtual address: 25 bits

$$32 \text{ Mb} = 33554432 \text{ bytes} = 2^{25} \text{ bytes}$$

A total of 25 bits are required to store the logical address.

b) physical address: 18 bits

$$256 \text{ Kb} = 262\,144 \text{ bytes} = 2^{18} \text{ bytes}$$

A total of 18 bits are required to store the physical address.

c) offset field (logical): 9 bits

page size = 512 bytes and therefore, offset can range from 0 to 511

2^9 = page size and the offset requires 9 bits for storage

d) page number field (logical): 16 bits

This is because the total bits for a logical address is 25 bits

minus the 9 bits for the offset.

e) number of page frames: 512 frames

$$\text{physical address space} = 256 \text{ Kb} = 262\,144 \text{ bytes}$$

$$\text{page size} = 512 \text{ bytes}$$

$$262\,144 / 512 = 512 \text{ frames}$$

M5. (15 marks)

(a) not done here.

(b) The question is: Given a virtual address written in binary as 0000 0000 1111 1111 1001 1001 1000 1000 or in hex as 00ff9988, and a page size of 256 (i.e., 2^8), what is the corresponding 24 bit physical address?

The last 8 bits of the 24 bit physical address will be the same as the last 8 bits of the virtual address, i.e., 1000 1000, because the page size is 2^8 . The first 16 bits of the physical address represent the frame number. Here, it is not possible to translate a virtual address to a physical address without some more information, namely which page frame corresponds to the specified page number. This information would be obtained from the page table. Suppose the page table contained an entry (in binary) relating

0000 0000 1111 1111 1001 1001 (page number) to 1011 1101 1110 0110
(frame number)

then the complete physical address is:

1011 1101 1110 0110 1000 1000

M7. a) At 90% hit ratio:

If page number is found in associative registers:

time required is 8 ns

If you fail to find the page number by looking in the associative register, then you also must access memory for the page table and finally access the desired byte in memory:

time required is 8 ns + 80 ns = 88 ns

Therefore, effective access time = $0.9 * 8 + 0.1 * 88 = 16$ ns

b) At 50% hit ratio:

effective access time = $0.5 * 8 + 0.5 * 88 = 48$ ns

c) The results from parts a) and b) suggest the following about the usefulness of associative registers for storing page-table entries:

- associative registers are most useful if the hit ratio is high

(Siberschatz & Galvin, OS Concepts, 6th edition, Sec., 9.4.2; 5th edition, Sec. 8.5.2.1)

M9 a) install a faster CPU:

This will not increase CPU utilization because as it is, the processor does not have enough work to keep it busy. It is only working 20% of the time. Therefore, increasing speed will get the jobs done faster but this does not matter because the processor does not have more work to do once its job is complete.

b) install a bigger paging disk:

Again, CPU utilization is not affected. A bigger disk will enable more data to be held on this busy disk. It does not affect speed of paging; there is simply a larger area to swap information from.

c) increase the degree of multiprogramming:

This will not improve CPU utilization. The ability to run more processes at once will exist. However, if this occurs, there is a need to do more switching among these processes. The paging disk is already very busy and with more page faults, things will slow down even more.

d) decrease the degree of multiprogramming:

This will increase the CPU utilization because there will be less switching of information from disk to main memory. The access of the disk is slower so less access will speed things up and produce more throughput.

e) install more main memory:

This will definitely increase CPU utilization. Access to main memory is much faster than access to disk. Therefore, the more data that can be stored in main memory, the faster processes are executed. Thus, there is increased throughput.

f) install a faster hard disk or multiple controllers with multiple hard disks:

Yes, this increases CPU utilization. This is because the paging disk is very busy. If this hard disk is faster, the swapping of information will be faster, providing for more work done in the same amount of time.

g) add prepaging to the page fetch algorithm:

This will decrease utilization of the CPU because some unnecessary pages will be suggested by the prepaging algorithm. Since prepaging requires unnecessary pages, the paging disk will remain busy and CPU utilization will not increase.

h) increase the page size:

The effect of an increased page size will depend on how the OS allocates the larger pages. If the page size is increased, the number of pages in main memory will be reduced.

The OS has two choices:

(1) keep the same number of pages per process, but reduce the number of processes in memory

(2) keep the same number of processes, but reduce the number of pages per process

In case 1, a process kept in main memory will have more room for its needed instructions and data in its new, larger pages. Thus, each process will require less paging and the CPU utilization will increase.

In case (2), a process will have fewer, larger pages. By the principles of locality, these will have less chance of providing needed instructions and data. Thus, each process will require more paging. Since the paging disk is already fully used, more paging cannot be provided and CPU utilization will decrease.

Fragmentation:

What kind of fragmentation do we have with paging? Given N processes in memory, how much fragmentation do we have?

Ans: with paging we have internal fragmentation only. Given N processes, we have $N * (\text{pagesize}/2)$ bytes of internal fragmentation.

What kind of fragmentation do we have with fixed partitions? Given N processes each in size M memory partitions how much fragmentation do we have?

Ans: with fixed partitions, we have mostly internal fragmentation. We could have some external fragmentation if the partitions cannot be merged and there is a large request. With N process of size M partitions, we have $N * (M/2)$ bytes of internal fragmentation.

Explain what happens in terms of the OS and the memory management unit when a process accesses an address that is out of range (such as, the famous "segmentation fault"). Use as much detail as you can.

Ans: the memory request is compared against a limit register (or equivalent bit in the page table hardware) and fails the test. It generates an interrupt. The OS, servicing the interrupt, checks the memory request and sees that it is outside the valid range (or to an invalid page). The OS then terminates the process (or whatever is appropriate).

Consider the following page reference string:

5, 2, 3, 7, 6, 3, 2, 5, 2, 3, 2

How many page faults would occur for an Second-Chance algorithm using 3 frames? Assume pure demand paging so your first unique pages will all cost one fault each.

Ans: 7 faults

Consider a demand-paging system with the following time-measured utilizations:

CPU utilization: 90%

Paging disk: 10%

Other I/O devices: 8%

Which (if any) of the following will (probably) decrease CPU utilization?

Install a faster paging disk

Increase the degree of multiprogramming

Decrease the degree of multiprogramming

Install more memory

Upgrade to a higher-bandwidth network

Ans: Basically, the CPU is fairly busy and the paging disk is fairly idle. Installing a faster disk won't help make the CPU less idle since page fault can just be serviced faster. Increasing the degree of multiprogramming will make more page faults, possibly lowering the utilization. Decreasing the degree of multiprogramming will probably decrease the number of page faults and keep the CPU even busier. Installing more memory will also probably increase CPU utilization. A high-bandwidth network would only increase the utilization of the other devices and increase CPU load more.

So, only (b).

What is the locality of reference? How does demand paging take advantage of this in terms of a process' working set?

Ans: The locality of reference is the notion that a process continues to execute in the same pages it has in the past. This is the locality. Localities can move and overlap, but at a given time there is typically a small set of active pages out of the total number of possible logical pages. The working set is the OS's representation of the process' locality. It is the set of recent page accesses and is meant to represent those pages that the process needs in memory at the current time.

Consider a paging system with the page table stored in memory:

If a paged memory reference takes 220 nanoseconds, how long does a memory reference take?

Ans: 110 nanoseconds

If we add associative registers, and we have an effective access time of 150 nanoseconds, what is the hit-ratio? Assume access to the associative registers takes 5 nanoseconds.

Ans: $150 = \text{hit} * (110+5) + (1-\text{hit}) * (110+5+110)$
 $\text{hit} = .68$, or 68%

Explain what happens when a process executes an `exec ()` call in terms of the paging environment.

Ans: the `exec()` call keeps the same process information (id, os resources, priority) but it loads a new program. To do this, it creates a new page table since the previous one is no longer valid. The new page table will have pages set up to the new text segment (the code executed) and a new stack and heap. All previously allocated frames can be reallocated.

What is the maximum number of disk I/O operations required to add a block to the end of the file a file descriptor of type:

linked-list Ans: needs to read every block first

linked-list allocation with index Ans: 1

i-nodes Ans: 3, in the worst case for a really big file (to read the blocks of pointers)

Assume the file descriptor and the data to be written is currently cached in memory, but nothing else is.

Explain the principles behind the "top half" of an interrupt handler and the "bottom half."

Ans: the top half handler does the minimum required to service the interrupt so interrupts can be re-enabled quickly. Typically, this minimum is to enqueue the bottom half handler that will do the bulk of the device driver work.

What is the role of the device independent layer of a device driver?

Ans: two fold. To provide a uniform layer to the applications so that they can invoke the same API on multiple devices. And to provide a mechanism for the device driver layer below to have services provided without adding their own system calls.

Explain why a Micro-Kernel architecture is, in theory, more robust than a Monolithic-Kernel architecture. Is Windows-NT a true Micro-Kernel? Explain why or why not.

Ans: Micro-kernel architectures operate by having a minimal kernel with other traditional OS services being done at the user level, outside the OS. Thus, if there is a bug in one of these services (say, a new file system has a bug in the code), the OS does not crash. Instead, the service can be restarted.

WindowsNT is not a pure micro-kernel because it has a "layered" approach to OS services with an "executive" layer handling some privileged OS services. Moreover, some traditional services, such as graphics rendering, were moved into the kernel for efficiency, but violate the micro-kernel principle more.

Select all relevant choices from the list below. The notion of "platform dependence" for various software packages comes from the fact that:

Machine architectures differ from computer to computer

Shells differ from OS to OS

System calls differ from OS to OS

Process scheduling algorithms differ from OS to OS

Ans: a) Because machine instructions are different you have to recompile a program in order to have it run on a different platform; and c) because system calls are different, access many OS services (which nearly all programs use) must be changed. Not b) because the shell, itself, can be ported and is not (typically) part of the OS; and not d) because programs do not (should not) rely upon the mechanics of a scheduling algorithm in order to run correctly.

Critical region access:

What is "busy waiting"?

Ans: using up CPU cycles (being "busy") while not able to gain access to a critical region ("waiting").

What is "blocking"?

Ans: where the OS suspends the process (puts it in the "waiting" or "blocking" state) when a process cannot access a critical region.

Consider the multiprocess SOS solution we looked at (`use-proc-table.c`). Explain why "busy waiting" is acceptable in order to gain access to the process control block.

Ans: In this case, the time spent busy waiting should be short. Plus, this code is being accessed when a CPU needs to pick another process to run. It is not clear what it would do anyway if it, say, "blocked" instead of busy-waited.

Why is it (often) faster to context switch between threads than it is to context switch between processes? Be specific. (Hint: see `system-thread.h`).

Ans: if the context-switch is between two threads belonging to the same process, then restoring the state may be quicker. In particular, the resources shared by the threads (the memory segments, primarily) do not need to be saved and restored.

In class, we noted that sometimes the `Dispatcher()` function causes the OS to execute `"while (1) { /* no op */ }"`. When and why does it do this?

Ans: It does this when it has no processes ready to run. The hardware always executes code, advancing the program counter, so the CPU has to be doing something. Moreover, not having this suggests the dispatcher() call would return suggesting the OS has ended ...

Consider compiling and running the following code on a Unix system:

```
#include <stdio.h>

int num;

int main() {
    num = 1;

    fork();

    num = num + 1;

    printf("%d\n", num);
}
```

```
}
```

What are the possible outputs?

Ans: "2 2". The "num" variable is not shared. You could also get just "2" if the fork call failed.

What if the `fork()` call was changed to a system called, say `spawn()` that created a new thread instead of a new process. Now, what are the possible outputs?

Ans: now, the "num" variable is shared. So, you can get "2 3" (threads execute serially), "3 2" (threads execute serially and the "2" thread is sliced out inside the `printf()` but before printing), "2 2" (the second thread's "+1" result is clobbered by the first thread), "3 3" (both threads add successfully, but the first is sliced out before printing)

Four processes are ready to run. Their CPU burst times are 10, 2, 4 and 1. In what order are they run under a SJF scheduling algorithm? What is the average waiting time? How close is this to the best average waiting time we could have had for these processes? What is the throughput?

Ans: SJF selects 1, 2, 4, 10

Average waiting time is: $(0 + 1 + 2 + 4) / 4 = 7/4$. This is the optimal waiting time since that is what SJF gives you.

Throughput is a rate, number of processes per time. There are 4 processes that complete in 17 units. So $4/17$.

True or False:

Monitors allow you to solve more types of synchronization problems than do semaphores.

Ans: False. monitors and semaphores have the same "expressive" power in terms of the synch problems they can solve. In fact, you can "build" a monitor with a semaphore and vice-versa.

`Test_and_Set()` and semaphores are essentially the same.

Ans: False. `Test_and_Set()` does busy waiting, semaphores do not. `Test_and_Set()` provides single access to a region, while semaphores can be initialized to numbers greater than 1.

"Busy waiting", even by the operating system, is always a big no-no.

Ans: False. The OS can busy wait if: it is the idle thread, it will be busy waiting only briefly, it has nothing "better" to do.

Semaphores always let 1 and only 1 process a time past after a `wait()` call.

Ans: False. You can initialize a semaphore to a number greater than 1. The value of the initial number is how many processes can "wait()" on the semaphore initially and not block.

In the dining philosopher's solution shown in class, suppose philosopher 2 was absentminded and never let go of his left fork, even while thinking. If all the other philosophers behaved normally and kept thinking and eating, what would be the eventual outcome of the system?

Ans: Eventually, 1 would grab his fork and block waiting for 2. 0 would do the same, waiting for 1 and so on around the table. Deadlock would be inevitable.

Suppose we tried to modify the Readers/Writers synchronization solution we saw in class to "favor" writers instead of readers:

Writer:

```
wait(mutex2);

writecount++;

signal(mutex2);

wait(wrt);

/* critical region */

signal(wrt);

wait(mutex2);

writecount--;

signal(mutex2);
```

Reader:

```
wait(mutex2);
```

```

if (writecount) wait(wrt);

wait(signal2);

wait(mutex);

readcount++;

if (readcount == 1) wait(wrt);

signal(mutex);

/* critical region */

wait(mutex);

readcount--;

if (readcount == 0) signal(wrt);

signal(mutex);

```

Does this work? Why or why not?

No, it does not work. Primarily, if writecount is greater than 0, the next writer will grab mutex2, then block on the wrt semaphore. This will cause deadlock as no other process (including the writer) will get a chance to go.

Consider your Project 1. Which of the following fields in the Linux struct task_struct (greatly condensed, here) could be used to solve the problem:

```

struct task_struct {

    volatile long state;    /* -1 unrunnable, 0
runnable, >0 stopped */

    long counter;

    long priority;

    int pid;

};

```

Briefly explain how they could be used.

counter is used as the dynamic priority. You can modify this according to how many processes the user has, relative to other users. For example, if a user has 3 processes and another has 1, both users would share a "total" counter value, but the first user would have its counter values set to $\text{total}/3$ and the second user would have its counter values set to $\text{total}/1$. There are probably other ways to solve this problem

Problem 10.1

When do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

Answer

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid a free frame is located and I/O requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the faulting instruction is restarted. Then the address translation does no longer result in a page fault.

Problem 10.3

A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

Answer

The virtual address in binary form is

0001 0001 0001 0010 0011 0100 0101 0110

Since the page size is 2^{12} (four kilo bytes), the page table has 2^{20} elements (slots). Therefore, the low-order 12 bits "0100 0101 0110" are used as displacement into the page, while the remaining 20 bits "0001 0001 0001 0010 0011" are used as the displacement into the page table.

Problem 10.4

Which of the following programming techniques and structures are "good" for a demand paging environment. Which are "not good"? Explain your answers.

Stack

Hashed Symbol Table

Sequential Search

Binary Search

Pure Code

Vector Operations

Indirections

Answer

Stack -- good, stack operations are local

Hashed Symbol Table -- not good, operations are not local

Sequential Search -- good

Binary Search -- not good, unless the table fits in few pages

Pure Code -- good, sequential access

Vector Operations -- good, sequential access

Indirections -- not good, contains jumps

Problem 10.5

Suppose we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty page is available or if a replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page fault rate for an effective access time of no more than 200 nanoseconds ?

Answer

$$0.2 \times 10^{-6} \text{ sec} \geq (1-p) \times 0.1 \times 10^{-6} \text{ sec} + 0.3 \times p \times 8 \times 10^{-3} \text{ sec}$$

$$+ 0.7 \times p \times 20 \times 10^{-3} \text{ sec}$$

$$0.1 \geq p(-0.1 + 2400 + 14000)$$

$$p \leq 0.000006$$

Problem 10.8

This problem is badly defined and cannot be solved with the values defined in the book.

Problem 10.9

Consider a demand paging system with the following time measured utilizations:

CPU: 20 %

Paging Disk: 97,7 %

Other I/O Devices: 5 %

Which of the following will probably improve CPU utilization ? Explain your answer.

Install a faster CPU

Install a bigger paging disk

Increase the number of processes

Decrease the number of processes

Install more main memory

Install a faster paging disk, or multiple controllers with multiple hard disk

Add prepaging to the page fetch algorithms

Increase the page size

Answer

Install a faster CPU -- No, CPU is waiting most of the time

Install a bigger paging disk -- No, it is not the problem

Increase the number of processes -- Never, will increase thrashing

Decrease the number of processes -- Best Idea

Install more main memory -- Why not? can hold more pages in memory and thus, obtain less page faults

Install a faster paging disk, or multiple controllers with multiple hard disk -- Yes, as the disk is the bottleneck, the CPU gets data more quickly.

Add prepaging to the page fetch algorithms -- Could help, above all, if programs follow the locality principle.

Increase the page size -- Will reduce the number of page faults if programs follow the locality principle. If not, it could result in higher paging activity because fewer pages can be kept in main memory and more data needs to be transferred per page fault.

Problem 10.10

Consider the two-dimensional array A:

```
int A[][] = new int[100][100];
```

where A[0][0] is at location 200, in a paged memory system with pages of size 200. A small process is in page zero (locations 0..199) for manipulating the matrix; thus, every instruction fetch will be from page zero.

For three page frames, how many page faults are generated by the following array initialization loops, using LRU replacement, and assuming page frame 1 has the process in it, and the two others are initially empty:

Solution a:

```
for (j=0; j < 100; j++)
```

```
    for (i=0; i < 100; i++)
```

```
        A[i][j] = 0;
```

Solution b:

```

for (i=0; i < 100; i++)

    for (j=0; j < 100; j++)

        A[i][j] = 0;

```

Answer

The array is stored row wise (row-major), that is, the first data page contains the elements $A[0,0], A[0,1], \dots, A[0,99], A[1,0], A[1,1], \dots, A[1,99]$, the second page contains $A[2,0], A[2,1], \dots, A[2,99], A[3,0], A[3,1], \dots, A[3,99]$, and so on.

Solution a:

The page reference string is

0,1,0,2,0,3,0,4,.....,0,49,0,50,0,1,0,2,0,3,.....0,50,.....

There will be $50 \times 100 = 5000$ page faults.

Solution b:

The page reference string is

0,1,0,2,0,3,0,4,.....,0,49,0,50

There will be $50 \times 1 = 50$ page faults.

Problem 10.11

Consider the following page reference string:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames ?

Remember that all frames are initially empty, so your first unique pages will all cost one page fault each.

LRU replacement

FIFO replacement

Optimal replacement

Consider the modified page reference string:

1112234444221115556222211122233777766333211122233366

How many page faults would occur for the working set algorithm if we use a window of (a) 6 references and (b) 10 references ?

Answer

Number of frames	LRU	FIFO	OPT
------------------	-----	------	-----

1	20	20	20
---	----	----	----

2	18	18	15
---	----	----	----

3	15	16	11
---	----	----	----

4	10	14	8
---	----	----	---

5	8	10	7
---	---	----	---

6	7	7	7
---	---	---	---

7	7	7	7
---	---	---	---

Working Set Algorithm

Window = 6

1112234444221115556222211122233777766333211122233366

1 1 11 2 21 1 1112 11 1 2 3363 211 1 1 22
 2 22 3 42 2 5525 22 2 3 7676 322 2 2 33
 33 4 4 5 656 6 3 7 7 7 633 3 6
 4 6 6

* * * * * * * * * *

17 page faults

Windows = 10

1112234444221115556222211122233777766333211122233366

1 1 11 1 11 111 1 2 3 21 1 1 1
 2 22 2 22 222 2 3 6 32 2 2 2
 33 4 45 6 3 3 6 7 63 3 3 3
 4 5 56 7 7 76 6 6
 6 7

* * * * * * * * *

12 page faults

Problem 10.17

Consider a demand paging system with a paging disk that has an average disk access time and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of one microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory, and that of the

remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time ?

Answer

effective access time =

0.8 1[us] + (associative hit ok)

0.2 (0.9 (1[us] + 1[us]) (associative miss)

0.1 (20[ms] + 1[us] + 1 [us]) (page transfer,memory access)

= 0.4 [ms]

Problem 6.1

A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many possible different schedules are there ? Give a formula in terms of n.

Answer

$n!$ (factorial of n) = $n * (n-1) * (n-2) * \dots * 3 * 2 * 1$)

Problem 6.2

Define the difference between preemptive and nonpreemptive (cooperative) scheduling. State why nonpreemptive scheduling is unlikely to be used in a computer center. In which computer systems nonpreemptive scheduling is appropriate?

Answer

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst. The danger with nonpreemptive scheduling is that a process may, because of programming mistake, enter an infinite loop, and then prevent the other processes to use the CPU. Nowadays, nonpreemptive scheduling is mostly used in small non time-critical embedded systems.

Problem 6.3

Consider the following set of processes, with the length of the CPU burst times given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, and P5 all at time 0.

draw four Gantt charts illustrating the execution of of theses processes using FCFS, SJF (equal burst length processes are scheduled in FCFS), a non-preemptive priority (small priority number means high priority, equal priority processes are scheduled in FCFS), and a RR (quantum=1) scheduling.

Answers

Time 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

FCFS: P1 P1 P1 P1 P1 P1 P1 P1 P1 P1 P1 P2 P3 P3 P4 P5 P5 P5
P5 P5

SJF: P2 P4 P3 P3 P5 P5 P5 P5 P5 P1 P1 P1 P1 P1 P1 P1 P1
P1 P1

PRIO: P2 P5 P5 P5 P5 P5 P1 P1 P1 P1 P1 P1 P1 P1 P1 P1 P1
P3 P4

RR: P1 P2 P3 P4 P5 P1 P3 P5 P1 P5 P1 P5 P1 P5 P1 P1 P1
P1 P1

What is the turnaround time of each process for each of the above scheduling algorithms
? What is the mean value ?

Answers

Turnaround time = waiting time + execution time

FCFS SJF PRIO RR

P1 10 19 16 19

P2 11 1 1 2

P3 13 4 18 7

P4 14 2 19 4

P5 19 9 6 14

Average 13.4 7.0 12.0 9.2

What is the waiting time of each process for of the above scheduling algorithms ?

Answers

Waiting time = turnaround time - burst time

FCFS SJF PRIO RR

P1 0 9 6 9

P2 10 0 0 1

P3 11 2 16 5

P4 13 1 18 3

P5 14 4 1 9

Average 9.6 3.2 8.2 5.4

Which of the schedules above results in the minimal average waiting time (over all processes) ? Why ?

Answer

Preemptive Shortest Job First has been proved to be optimal. However, in the reality it is impossible to know the exact CPU bursts in advance. One approach is to estimate the next CPU burst in terms of the history of the previous CPU bursts (cf. "exponential average" prediction).

Problem 3.1

What are the five major activities of an operating system in regard to process management ?

Solution

The creation and deletion of processes: both user and system processes

The suspension and resumption of processes

The provision of mechanism for process synchronization

The provision of mechanism for inter process communication

The provision of mechanism for deadlock handling

Problem 3.2

What are the three major activities of an operating system in regard to memory management ?

Solution

Keep track of which parts of memory are currently being used and by whom

Assign a portion of memory to newly created processes

Allocate and deallocate memory space dynamically as needed

Problem 3.5

What is the purpose of a command interpreter ? Why is it usually separate from the kernel ?

Solution

It reads commands from the user or from a file of commands (Unix shell script) and executes them, usually by turning them into one or more system calls.

A command interpreter is usually not part of the kernel because the command interpreter is subject to change.

Problem 3.7

What is the purpose of a system call ?

Solution

System calls allow user-level processes to request service of the operating system.

Problem 3.11

What is the main advantage of the layered approach to systems design ?

Solution

As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and to modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

Problem 3.12

What is the main advantage of the micro kernel approach to systems design ?

Solution

Benefits typically include the following points:

Adding a new service to the system does not require modifying kernel

It is more secure as more operations are done in user space than in kernel mode

A simpler kernel design and functionality typically results in a more reliable operating system

Problem 4.1

Several popular single-user microcomputer operating systems such as MS-DOS provide little or no means of concurrent processing. Discuss the major complications that concurrent processing adds to an operating system.

Answer

A method of time sharing must be implemented to allow each user process to have access to the system. This method is based on preemption of a process that does not voluntarily give up the processor (tries to monopolize it). The kernel must be re-entrant, so more processes may be executing kernel code concurrently.

Processes and system resources must be protected from each other. Any given process must be limited in the amount of memory it can use. The system resources such as files and devices must be protected from direct access from a user process (Bypass the operating system must not be possible).

Care should be taken to prevent processes from deadlocks.

Problem 4.2

Describe the differences among short-term, medium-term, and long-term scheduling.

Answer

Short term scheduler (CPU scheduler): selects a process from a set of processes in ready state (from ready queue) and allocates the CPU to it (dispatching).

Long term scheduler (job scheduler): determines which job from the system's job queue is brought into memory for processing. This scheduler makes only sense if the machine supports batch processing. No machine at the HTA Bienne does so.

Medium term scheduler (swapper): used especially in time sharing systems to swap a processes out and back if it becomes temporarily inactive, for example, if the user leaves a machine for a coffee break.

Problem 4.4

Describe the actions taken by a kernel to context switch between processes.

Answer

See textbook p. 98. figure 4.3

Problems of OSC

Problem 9.2

Explain the difference between internal and external fragmentation.

Answer Internal Fragmentation is the area in a region or a page that is not used by the job occupying that region or page. This space is unavailable for use by the system until that job is finished and the page or region is released.

Problem 9.3

Describe the following allocation algorithms:

First fit

Best fit

Worst fit

Answer

First-fit: search the list of available memory and allocate the first block that is big enough.

Best-fit: search the entire list of available memory and allocate the smallest block that is big enough.

Worst-fit: search the entire list of available memory and allocate the largest block. (The justification for this scheme is that the leftover block produced would be larger and potentially more useful than that produced by the best-fit approach.)

Problem 9.5

Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

Answer

First-fit:

212K is put in 500K partition

417K is put in 600K partition

112K is put in 288K partition (new partition $288K = 500K - 212K$)

426K must wait

Best-fit:

212K is put in 300K partition

417K is put in 500K partition

112K is put in 200K partition

426K is put in 600K partition

Worst-fit:

212K is put in 600K partition

417K is put in 500K partition

112K is put in 388K partition

426K must wait

In this example, Best-fit turns out to be the best.

Problem 9.7

Why are page sizes always powers of 2?

Answer Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

Problem 9.8

Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.

How many bits are there in the logical address?

How many bits are there in the physical address?

Answer

Logical address : 13 bits

Physical address : 15 bits

Problem 9.9

On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?

Answer

An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. There is no way for a process to refer to a page it does not own because the page will not be in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process's page table. This is useful when two or more processes need to exchange data they just read and write to the same physical addresses (which may be at varying logical addresses). This makes for very efficient interprocess communication.

Problem 9.10

Consider a paging system with the page table stored in memory.

If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)

Answer

400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

Effective access time = $0.75 * (200 \text{ nanoseconds}) + 0.25 * (400 \text{ nanoseconds}) = 250$ nanoseconds.

Problem 9.11

What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on the one page have on the other page

Answer: By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. "Copying" large amounts of memory could be effected by having different page tables point to the same memory location.

However, sharing of nonreentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user's "copy."

Problem 9.12

Why segmentation and paging sometimes combined into one scheme?

Answer:

Segmentation and paging are often combined in order to improve upon each other. Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single segment table entry with a page table address of zero. Pages segmentation handles the case of having very long segments that require a lot of time for allocation. By paging the segments, we reduce wasted memory due to external fragmentation as well as simplify the allocation.

Problem 9.13

Describe a mechanism by which one segment could belong to the address space of two different processes.

Answer:

Since segment tables are a collection of base-limit registers, segments can be shared when entries in the segment table of two different jobs point to the same physical location. The two segment tables must have identical base pointers and the shared segment number must be the same in the two processes.

Problem 9.14

Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.

Answer:

Since segmentation is based on a logical division of memory rather than a physical one, segments of any size can be shared with only one entry in the segment tables of each user. With paging there must be a common entry in the page tables for each page that is shared.

Problem 9.16

Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

- a. 0,430
- b. 1,10
- c. 2,500
- d. 3,400
- e. 4,112

Answer:

- a. $219 + 430 = 649$
- b. $2300 + 10 = 2310$
- d. illegal reference, trap to operating system
- d. $1327 + 400 = 1727$
- e. illegal reference, trap to operating system

Problem 9.17

Consider the Intel address translation scheme shown in Figure 9.20 in OSC (page 292).

Describe all the steps that are taken by the Intel 80386 in translating a logical address into a physical address.

What are the advantages to the operating system of hardware that provide such complicated memory translation hardware?

Are there any disadvantages to this address translation system?

Answer:

a. The selector is an index into the segment descriptor table. The segment descriptor result plus the original offset is used to produce a linear address with a dir, page, and offset. The dir is an index into a page directory. The entry from the page directory selects the page table, and the page field is an index into the page table. The entry from the page table, plus the offset, is the physical address.

b. Such a page translation mechanism offers the flexibility to allow most operating system to implement their memory scheme in hardware, instead of having to implement some parts in hardware and some in software. Because it can be done in hardware it is more efficient (and the kernel is simpler).

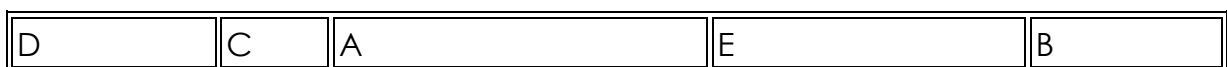
c. Address translation can take longer due to the multiple table lookups it can invoke. Caches help, but there will still be cache misses.

-->

5. Five jobs A through E arrive at a computer center having estimated run times of 10, 6, 2, 4, and 8 minutes. Their priorities are 3, 5, 2, 1, and 4, respectively (with 1 being the highest priority). For each of the following scheduling algorithms, plot the Gantt chart and determine the mean process turnaround time.

(a) Priority scheduling.

Gantt chart is plotted as follows:



0 4 6 16 24 30

The mean process turnaround time = $(4 + 6 + 16 + 24 + 30) / 5 = 16$ minutes/process

(b) Round robin.

Assume that time quantum is 1 minute.

The Gantt chart is plotted as follows:

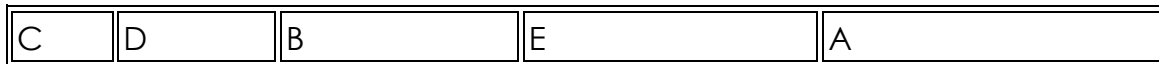


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The mean process turnaround time = $(30 + 23 + 8 + 17 + 28) / 5 = 21.2$ minutes/process

(c) SJF.

The Gantt chart is plotted as follows:



0 2 6 12 20 30

The mean process turnaround time = $(2 + 6 + 12 + 20 + 30) / 5 = 14$ minutes/process

6. Assume a program with eight virtual pages, numbered from 0 to 7. The pages are referenced in the order of 0123012301234567. Show how many page faults will happen (by pointing out where cause these page faults) if you use a three page frames and the following strategies:

(a) Least Recently Used(LRU).

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

There are 16 page faults.

(b) Least Frequently Used(LFU).

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

There are 16 page faults.

(c) First-In First-Out(FIFO).

0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

There are 16 page faults.

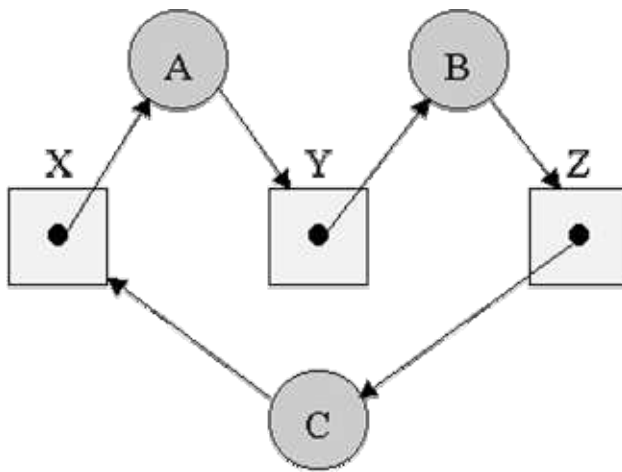
7. Given three processes A, B, and C; three resources X, Y, and Z; and the following events: (1) A requests X, (2) A requests Y, (3) B requests Y, (4) B requests Z, (5) C requests Z, (6) C requests X. Assume that the requested resource should always be allocated to the request process if it is available.

Draw the resource allocation graph for the following sequences

respectively, and tell whether it is a deadlock? If it is, how to recover?

(a) Events occur in the sequence of 1, 3, 5, 2, 6, and 4.

The resource allocation graph is shown as follows:

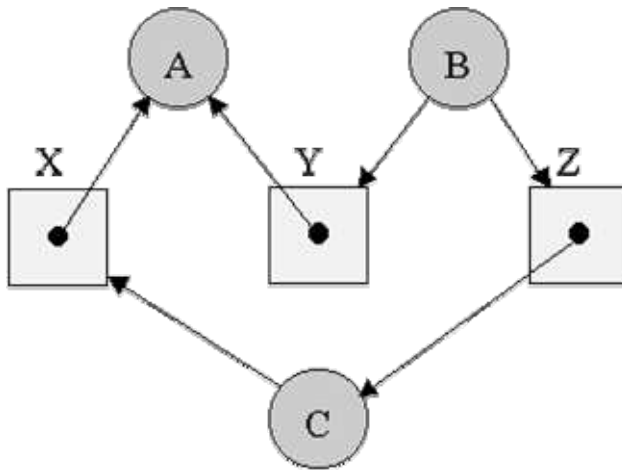


Because each resource type has only one instance, and there is a cycle in the resource allocation graph, we can see that a deadlock has occurred.

There are two methods to recover from a deadlock. One is process termination, the other is resource preemption. For example, we choose the former method, we can choose one process to be terminated. If we choose process B to be terminated, then resource Y can now be allocated to process A, and process A can successfully finish its job and finally release both resource X and Y, so that we further allocate resource X to process C, and now process C can successfully finish its job.

(b) Events occur in the sequence of 1, 5, 2, 6, 3, and 4.

The resource allocation graph is shown as follows:



Because there are no cycles in the resource allocation graph, we can see that no process in the system is deadlocked.

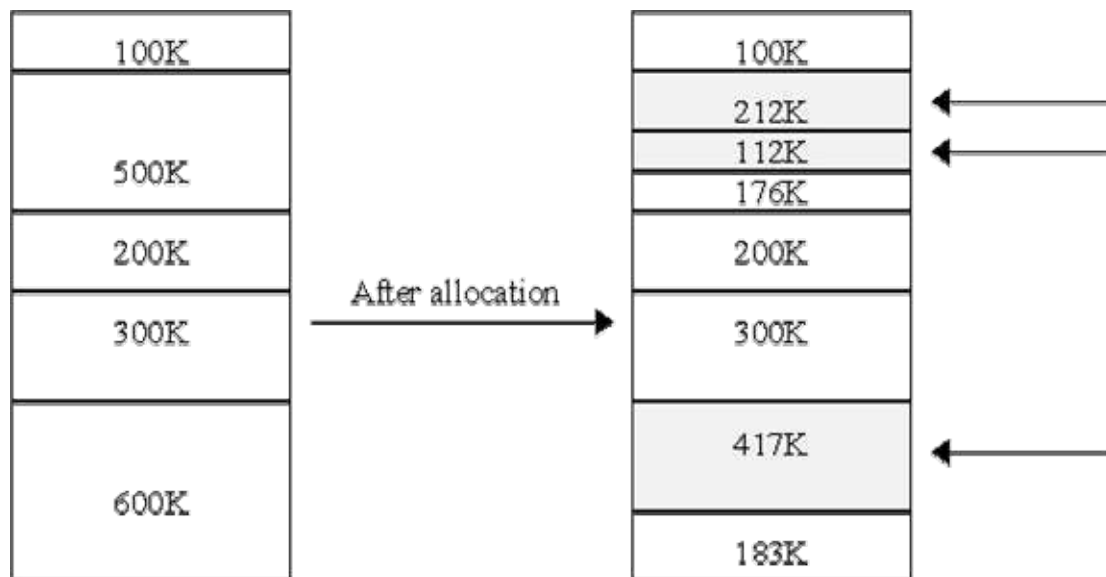
Chapter 8

Exercise 8.5

Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

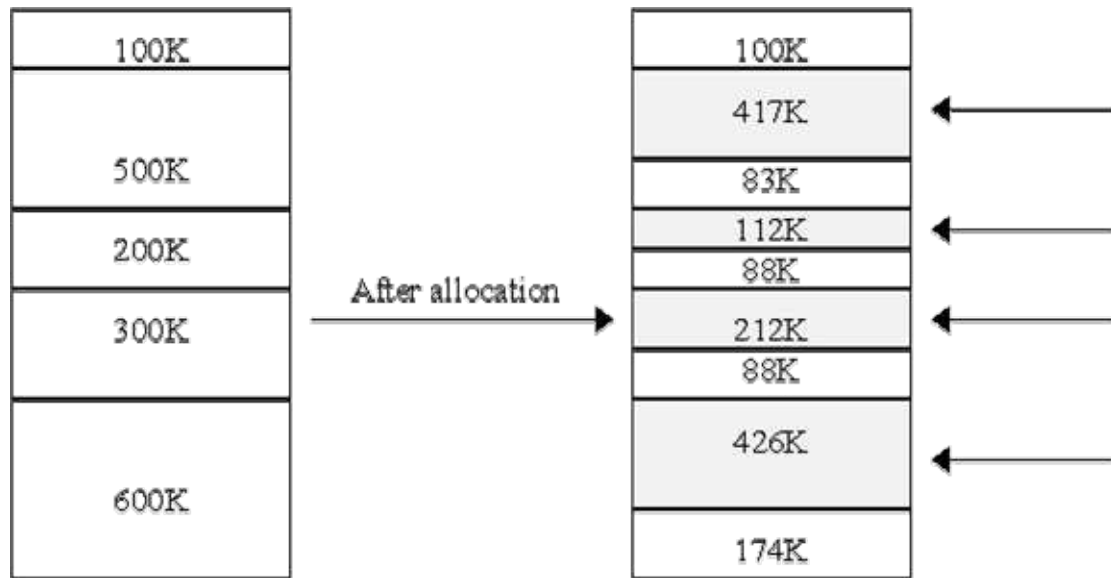


For the First-fit algorithm:

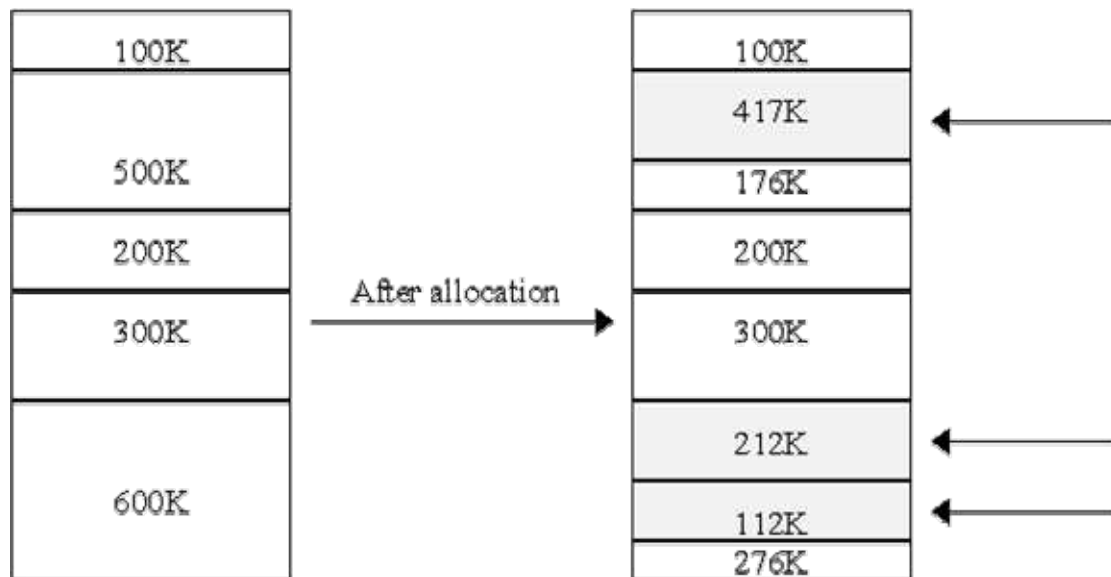


The request of memory allocation for 426K will be rejected, because any single partition in the graph is less than 426K.

For the Best-fit algorithm:



For the Worst-fit algorithm:



The request of memory allocation for 426K will be rejected, because any single partition in the graph is less than 426K.

Finally, we can see that the Best-fit algorithm can fulfill all four requests for memory allocation, so it's the most efficient one!

Exercise 8.16

 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430 / b. 1,10 / c. 2,500 / d. 3,400 / e. 4,112



a. The physical address is $219 + 430 = 649$

b. The physical address is $2300 + 10 = 2310$

c. The offset in the logical address is larger than the length of the segment corresponding to that logical address ($500 > 100$), so, this access to memory is illegal. An error has occurred.

d. The physical address is $1327 + 400 = 1727$

e. The offset in the logical address is larger than the length of the segment corresponding to that logical address ($112 > 96$), so, this access to memory is illegal. An error has occurred.

Exercise 8.17

 Consider the Intel address translation scheme shown in Figure 8.28.

- a. Describe all the steps that are taken by the Intel 80386 in translating a logical address into a physical address.
- b. What are the advantages to the operating system of hardware that provides such complicated memory translation hardware?
- c. Are there any disadvantages to this address translation system?



a.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

s	g	p
13 bits	1 bit	2 bits

in which s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte (word) within the segment in question.

The physical address on the 386 is 32 bits long and is formed as follows. The select register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question are used to generate a linear address. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.

The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows:

page number		page offset
p1	p2	d
10 bits	10 bits	12 bits

p1 points to the appropriate entry in the page directory. Together with page directory base register, we can find the base address of the page table. p2 points to the appropriate entry in the page table. Together with the base address of the page table, we can find the corresponding page frame in physical memory. Adding this to the page offset, finally we can find the physical address in question.

b.

At first, the memory is segmented, with the following advantages:

1. It lends itself to protection. Because the segments represent a semantically defined portion of the program, it is likely that all entries in the segment will be used the same way. Thus simplifies the task of protection.
2. It lends itself to sharing. Each process has a segment table associated with its process control block, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point to the same physical locations.

Then, each segment is paged, with the following advantages:

1. A segment may be broken up into a number of pieces and these pieces need not be contiguously located in main memory during execution.

2. With two-level paging, the size of the page table is smaller.

3. There's no external fragmentation.

c.


Disadvantages to this address translation system:

1. Each access to memory will require four references to the memory (three references to locate the appropriate entry in the descriptor table, the page directory, the page table, respectively, and one to access to the content in the resulting physical address). So, the time required to map a logical address to a physical address increases. However, a set of associative registers can reduce the performance degradation to an acceptable level.

2. With paging, there is the problem of internal fragmentation.

Chapter 9

Exercise 9.9

 Consider a demand-paging system with the following time-measured utilizations:

Paging disk	97.7%
Other I/O devices	5%

Which (if any) of the following will (probably) improve CPU utilization?
Explain your answer.

- a. Install a faster CPU.
- b. Install a bigger paging disk.
- c. Increase the degree of multiprogramming.
- d. Decrease the degree of multiprogramming.
- e. Install more main memory.
- f. Install a faster hard disk, or multiple controllers with multiple hard disks.
- g. Add prepaging to the page fetch algorithms.
- h. Increase the page size.



a. Because the CPU utilization is low (20%) and the utilization of the paging disk is very high (97%), we can see that the system is lack of free memory (too many processes are entered in the system at the same time), so merely install a faster CPU will not improve CPU utilization too much.

b. Because the system is lack of free memory, merely install a bigger paging disk may not be of much help, either.

c. Increase the degree of multiprogramming will further degrade the performance, because more processes have to be swapped in and out from memory now, the system will spend too much time in swapping than in calculation.

d. Decrease the degree of multiprogramming will be helpful, because it will decrease the frequency of swapping, leaving the system more time doing calculations.

e. Install more main memory will be helpful, because when there is more free memory, the frequency of swapping will decrease, leaving the system more time doing calculations.

f. Install a faster hard disk, or multiple controllers with multiple hard disks may be helpful, since now the system spends less time in swapping, it has more time doing calculations.

g. Add prepaging to the page fetch algorithms may be helpful or not. It depends on whether the cost of prepaging is less than the cost of servicing the corresponding page faults.

h. Increase the page size will degrade the performance, because the internal fragmentation will be worse, the utilization of main memory will be low, more processes will not be able to fit into main memory, and the system will have to spend more time in swapping.

Exercise 9.11



Consider the following page reference string:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement



	1 frame	2 frames	3 frames	4 frames	5 frames	6 frames	7 frames
LRU	20	18	15	10	8	7	7
FIFO	20	18	16	14	10	10	7
Optimal	20	15	11	8	7	7	7

Exercise 9.20

📖 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?



1. The cause of thrashing:

In the steady state, practically all the main memory will be occupied with process pieces, so that the processor and operating system will have direct access to as many processes as possible. Thus, when the operating system brings one piece in, it must throw another out. If it throws out a piece just before it is about to be used, then it will just have to go get that piece again almost immediately. Too much of this leads to a condition known as thrashing: The processor spends most of its time swapping pieces rather than executing user instructions.

2. How to detect thrashing:

We can measure the CPU utilization and the degree of multiprogramming at some interval, and see that whether while increasing the degree of multiprogramming introduces, the CPU utilization is also increased. If it is the case, there's no thrashing. Otherwise, there is thrashing.

3. Having detected thrashing, what can the system do to eliminate this problem:

The system can throw some of the processes out from main memory, and take them from the ready queue into the job queue for a while. Leave more room for other processes to run. This will eliminate the need for other processes to repeatedly and frequently swap in and swap out, thus will eliminate the problem of thrashing.

Q1: The first assignment has been released - some time has been reserved for discussing it. Note: Your tutor will not answer questions on it after the tutorial.

Q2: A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in cache, 20 ns are required to access it. If it is in main memory but not in cache, 60ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60ns to copy it to the cache, and then the reference is started again.

The cache hit ratio is 0.9 and the main memory hit ratio is 0.6. What is the average time in nanoseconds required to access a referenced word on the system?

$$0.9 * 20\text{ns} + (0.1 * 0.6) * 80 \text{ ns} + 0.04 * 12\text{ms} = 480022.80$$

Q3: Discuss the differences between programmed I/O, DMA and memory-mapped I/O. What devices are appropriate for each type?

Q4: Which is faster? Interrupts or polling? Under what circumstances?

Depends on the circumstances. Things to consider; Speed of devices. Frequency of interrupts.

Questions and Answers

Q1:

VPN	PFN	Loaded	Ref Time	Ref Bit
2	0	60	161	0
1	1	130	160	0
0	2	26	162	1
3	3	20	163	1

A fault to VPN 4 has occurred; which frame will be replaced on:

FIFO

LRU

Clock - assume the hand is at from 3

Optimal - future references are: 4,0,0,0,2,4,2,1,0,3,2

FIFO: 3, LRU: 1, Clock: 2, Optimal: 3

Assuming the reference string given in optimal, rank the algorithms in terms of best to worst in terms of performance (ie least page faults)

Q2: Some Operating systems discard program text pages and reload them from the original program binary. What are the advantages and disadvantages compared to paging to swap.

Adv:

Less swap used

Don't have to write binaries to swap

Paging system now relies on file system

May require binaries to stay in place while they run

May incur extra load time penalties

More complex paging system

Q3: Consider a demand-paging system with the following time-measured utilizations:

CPU utilization 20%

Paging disk 97.7%

Other I/O devices 5%

Which (if any) of the following will (probably) improve CPU utilization? Explain your answer.

Install a faster CPU.

Install a bigger paging disk.

Increase the degree of multiprogramming.

Decrease the degree of multiprogramming.

Install more main memory.

Install a faster hard disk, or multiple controllers with multiple hard disks.

Add prepaging to the page fetch algorithms.

Increase the page size.

Things that will help are: Less Multiprogramming, More memory (the best thing to do), faster hard disk.

Prepaging and page size increase may help or hinder depending on page fault locality.

Bigger disk will do nothing.

Faster CPU or more programs will make things worse.

Solutions to exercises from the textbook (Silberschatz et al., 2000):

6.1 A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many possible different schedules are there? Give a formula in terms of n .

For 1 process, there is only one possible ordering: (1). For 2 processes, there are 2 possible orderings: (1, 2), (2, 1). For 3 processes, there are 6 possible orderings: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). For 4 processes, there are 24 possible orderings. In general, for n processes, there are $n!$ possible orderings.

6.7 Consider the following preemptive priority-scheduling algorithm that is based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α ; when it is running, its priority changes at a rate β . All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

What is the algorithm that results from $\beta > \alpha > 0$?

What is the algorithm that results from $\alpha < \beta < 0$?

Since processes start out with priority 0, any processes already in the system (either running or waiting) are favoured. Therefore, new processes go to the back of the queue. Running processes' priorities increase at a greater rate than those of waiting processes, so they will never be preempted. Thus, we have FCFS.

This time, process priority decreases the longer a process has been in the system (either running or waiting). New processes start out at 0, and thus have a higher priority than processes that have already been in the system. The scheduler will preempt in favour of newer processes. In the absence of new arrivals, the scheduler will pick from the ready queue the process that has been waiting the least amount of time, since priority decreases faster waiting than running. We therefore have a LIFO (last in, first out) scheduling algorithm.

10.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p ; n distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:

What is a lower bound on the number of page faults?

What is an upper bound on the number of page faults?

There are n distinct pages occurring in the reference string, each of which must be loaded into memory at some point. Therefore the lower bound on the number of page faults is n , which is irrespective of the memory size, m .

The upper bound depends on the memory size, m . If $m \geq n$, then the upper bound will be n irrespective of p , since all the pages referenced in the string can be loaded into memory simultaneously. If $m = 1$, then the upper bound is 1 when $n = 1$, and p when $n > 1$. Similarly, for other values of m :

p	n	Upper bound				
		m=1	m=2	m=3	...	m=n
1	1	1	1	1	...	1
2	1	1	1	1	...	1
2	2	2	2	2	...	2
3	1	1	1	1	...	1
3	2	3	2	2	...	2
3	3	3	3	3	...	3
4	1	1	1	1	...	1
4	2	4	2	2	...	2
4	3	4	4	3	...	3
4	4	4	4	4	...	4
5	1	1	1	1	...	1
5	2	5	2	2	...	2
5	3	5	5	3	...	3
5	4	5	5	5	...	4
5	5	5	5	5	...	5

Thus it can be seen that the upper bound is n when $m \geq n$, and p when $m < n$. If you have trouble seeing this, it may help to go sketch a few page tables by hand for successive small values of m .

10.9 Consider a demand-paging system with the following time-measured utilizations:

CPU utilization: 20%

Paging disk: 97.7%

Other I/O devices: 5%

For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.

install a faster CPU

install a bigger paging disk

increase the degree of multiprogramming

decrease the degree of multiprogramming

install more main memory

install a faster hard disk, or multiple controllers with multiple hard disks

add prepaging to the page-fetch algorithms

increase the page size

This will probably not increase CPU utilization — on the contrary; it will probably decrease it. The given data suggest that most of the time the CPU is idling, waiting for the paging disk to complete its task. Since the CPU is dependent on the paging disk, installing a faster CPU will only cause processes to execute more quickly and place a greater demand on the paging disk.

This will probably not increase CPU utilization. A frequently-accessed paging disk, as in this example, does not imply that the disk is too small; it implies that memory required for the number of running processes is inadequate. A larger disk might even decrease CPU utilization, since more time will be spent performing seeks to the hard drive.

Of course, this will increase CPU utilization in general, but it will decrease CPU utilization per unit time. A frequently-accessed paging disk implies that there is not enough physical memory for the number of running processes, so by adding more processes, we are only decreasing the amount of memory available to each process and thus increasing the paging disk usage. This means more CPU time spent waiting for the paging disk and less time spent performing useful work.

This will increase the amount of CPU usage per unit time. Because the paging disk is saturated with requests, this implies that RAM is inadequate for the number of processes currently running. Thus, if we were to decrease the number of processes running, it would make more physical memory available to each process, so there would be fewer chances that the processes would page fault. Fewer page faults mean less paging disk accesses, which means less time spent by the CPU waiting for disk I/O.

This will definitely increase the amount of CPU usage. Since more memory is available to each process, more pages for each process can be in RAM simultaneously, and there is a lower probability that any given process will page fault. As stated before, fewer page

faults mean fewer paging disk accesses, which means less time spent by the CPU waiting for the paging disk.

This will improve CPU utilization, though probably not as much as the previous two options. Increasing the paging disk speed will result in less time spent by the CPU waiting for the paging disk to complete its tasks, so it can spend more time executing processes.

Depending on the processes, this may improve CPU utilization. If a large proportion of the page faults is being caused by a process that pages in several frequently-used pages during every one of its time slices, then adding a prepaging algorithm will cut down on some disk latency time by reading in most or all of the necessary pages at once, rather than waiting for the process to page fault for each page. Fewer page faults mean less idle CPU time.

Again, this depends on the processes, but it may improve CPU utilization. If processes are continually page-faulting on logically adjacent pages, then increasing the page size will result in fewer page faults, and therefore less disk access time and less CPU idling. However, if this is not the case, then increasing the page size will only worsen the problem, since more time will be spent by the paging disk reading in data or code that will not be used anyway.

Abhishek Pharkya