OS Introduction Most computers have two modes of operation: kernel mode and user mode. The operating system is the most fundamental piece of soft ware and runs in kernel mode (or supervisor model.

In this mode it has complete access to all the hardware and can execute any instruction the machine is capable of executing. The rest of the soft ware runs in user mode, in which only a subset of the machine instructions is available. In particular those instructions that afect contol of the machine or do Ilo ove forbidden to user-mode programs. The user interface program, shell or Gul, is the lowest level of user-mode software, and allows the user to start other

ser to start other programs (web browar). Os nous on bare hardware and

programs & web provides the base for all other software.

An important distination between the operating system and normal (uver mode) software is that if a user does not like a particular e-mail realer, she is free to get I write another one; (she is not free to write their own clock intempt handler, which is part of the os and is protected by hardware against attempts by users to mootty" The distinetion is some times blurred in embedded on interpreted systems (such as Sara - bared os that use interpretation, not hardware to separate the components. Everything woning in kernel mode is clearly rout of the Os, but some programs running outside it are arguably also part of it, are closely associated with it like File Sy tems & password changing. What is an OS?

Softwere which was in kamel mode. It basically performs two perelated functions providing application programmers (and application programmers, naturally) a clean as

amers naturally) a clean abstract set of resunes instead of the messer hardwere ones and managing there hardware resources, The job of the OS is to create good abstractions and then implement and manage the abstract objects this created. One of the major tasks of the op OS is to hide the hardware and present mograms (and their programmes with nice clean, elegant, consistent, abstractions to work with insteal. Os's real customers are the application programs (via the application programmers).

They are the ones who deal clivectly with the os and its abstractions. In contrast end uses deal with the abstractions provided by the user interface aul/ shell. In an alternative (bottom-up) view, the job of the Os is to

the io6 of than os is to mauide

provide for

for an orderter

an orderly and contro

and controlled allocation of the process memories, and I lo devices among the various progans competing for them. In short, this view of the as holds that its primary task is to keep track of which programs are using which resource, to grant resource requests to account for usage, and to mediate conftretiny requests from different mograms and users. Resances are sherred in two ways time & space.

Computer Hardware review Program counter contains the memory address of the next instruction to be fetched. Alter the instruction has been fetched, the program

cram counter is uocated to point to its

counter is updated to point.

successor.

current stack Stadl pointer points to the top of the

one frame

in memory. The stack contains for each procedure that has been entered but not yet exited. A procedure's stack frame holds those input parameters, local variables, and temporary veritables that are not kept in registers. Most Os's keep pieces of) (cache) heavily used files in main memory to avoid hewing to fetch them from the disk repeatedly

*Il*0 Devices Ilocan be done in 3 ways. One (simplest) a user program issues a system call, which the Kernel then translates into a procedure call to the appropriate driver. The driver then starts the 110 and sits in a tight loop continuously polling the device to see if it is done. When Ilo is done, the driver puts the data (it any) where they are needed and returns. The os then returns control to the caller. This method is called busy waiting and has the clistadvantys of tying up the CPU polling the device until it is finished. The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that point the driver returns. The os then blocks the calles is needed and looks for other work to do. When the controller detects the end of the transfor, it generates an interrupt to signal completion. 3rd way use a DMA chip (Direct memory Access).

Os concepts - Processes In many oss, all the information about each process, other than the contents of its own address space, is stored in an os table called the press table which is an array ( linked list) of structures, one for each process currently in existence.

System Calls Making a system call is like making a special keind of procedure call, only system calls enter the kernel and procedure calls do not os structure There are around six different os structures including monolithic systems, kagered systems, microkernels client server systems, virtual machines and exo kenels. Monolithic Systems

In this approach the entire os runs as a single program in Icenel mode. The OS is written as a collection of mocedures, linked together into a single large executable binary proppam.

Each mocedure in the system is free to cals any other one. Having thousands of rocedures that can call each other without restriction often leas to an unwielelly and difficult to understand system There is no information hiding, every procedure is visible to every other procedure, Basic structure of os:

A main program that invokes the requested service procedure

• A set of service mocectures that carry out the system cells.

It set of utility procedures that help the service procedures. Layered Systems

anise the OS as a hierarchy of layes, each one constructed from the one below it.

Mieno kernels

The basic idea behind the microkemel design is to achieve high reliability by splitting the Os up into small, well-defined modules, only one of which the micro kernel runs in leemel mode and the pest on as relatively powerless ordinary user processes. Safar as a Gug in the kened can bring down the system instantly. About to ougs por 1000 lines of code.

switching back and for

differences between a process and a mas a program, input, output aure

Processes and threads

Process: an abstraction of a wnning program. Then support the ability to have even wh*e*n *th*ere is only ore CPU available. T*he*y tum a single CPU into multiple virtual CPUs.

(pseudo) concurrent operators

Processes The Process M*o*del A process is just an instance of an execting program, including the wreat values of the program counter recnisters and variables. Conceptually each process has its ours virtual CPU. In reality of course the real CPU switches back and forth from process to process, out to understand the system, it is much easier to think about a collection of mocesses winning in (pseudo) parall than to try to keep track of how the CPO switches from mogram to program. This rapid switching back and forth is called multi programming č po switches rupilly back and forth among the processes. The ditterence between a process and a mogram is subtle, but critical

The key idea is that a process is an actrity of some kind. It has a program, input, output and astate. Ida mogram is running twice, it *co*unts as *t*wo processes.

Process Creation There are four principal events that cause processes to be created :

System initialisation 2) Ereation of a process creation system call og a running process. 8) Auser request to create a new process 4) Initiation of a batch gob. Processes that stay in the background to handle some activity such as e-mail, web pages, news, printing and so on are called daemons. In Only there is only one system call to weaté a new process: fork. This call creates an exact clone of the calling process. After the forks the two processes the parent and the child, have the same memory image the same environset strings, and the same open open files. Usually, the child process then executes execue or a similar

and on a new program. In both unlix and windows, ystem call to change its memory imate efter a process is created, the parent and child have their own distinct address spaces.

other process, If either process chance a word in its address space, the change is not visible to the In Unix, the child's initial address space is a copy of the parent's, but there are definitely two distinct address spaces involved; no writable memory is sherred. It is however possible for a newly created process to share some of its creators other resources, such as open files. In Windows, the parent's and child's address spaces are different from the start.

Process Termination

A process terminutes due to one of the following conditions. 1) Normal exit (voluntary) 2) Error exit (voluntary) (program bug) 3) Fatal exit (involuntary) (bg handler) 4) killed by another process (involuntary)

In some systeas, when a process derminates, either voluntarily or otherwise all mocesses it created are immediately killed as well. Neither unix non Windows. work this way, however.

*P*

Process Hrerarchies.

In some systems, when a process creates another, process, the parent mocess and child process continue to be associated in certain ways. The child process can it self create more processes, forming a process hierachry. In UNIX a process and allofits children, together form a process group. Windows does not have a moress herradry, Processes in UNIX cannot dis inhen't their children.

Process States

(Running)

Althe ich moress is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input. A mocess can be in one of three states:

Running Cactually uring the CPU out that instant) 2) Really (runnable; *t*emporarily stop*p*eal to let another process nn) *3*) 13 locked (unable to run until some external event happens). hogically the first two states are similar. In Goth cases the process is willing to run, only in the second one, there is tem porarily, no CPU available for it. *The th*ird state is different from *the*

first two in that the process cannot run even if the CPU has nothing else to do.

Iransition & occurs when the external event for which a

**1**

process was waiting Csuch as the arrival of some input) happens. If no other process is nnning at that instant,

transition *3 will b*e triggered are the process will start running. Otherwise it may have to wait in ready state for a

little while until the CPU is available and ist's turn comes.

Transitions between states.

The lowest level of the Os is the scheduler, with a variety 1) Process blocks for input of processes on top of it. All the interrupt handling and 2) Scheduler picks another process details of actually starting and stopping processes are 3) Scheduler picks this process hidden away in the scheduler, which is actually not too much H) Input becomes available. cocle. The rest of the OS is nicely structured in process form. Few

realt systems are as nicely structured as this

ol11.... n-1

(Blocked)

(Ready

processes

ier

Implementation of processes

I sheduler

To implement the process model, the Os maintains a table (an array of structures), called the process table, with one entry per process*. This* entry contains important information about the process a state, including its program counter, stack pointer memory allocation, the status of its open files its accounting and scheduling information and everything else about the proces that must be saved when the process is switched from running to ready on blocheed state so that it can be restarted later as if it had never been stopped. Associated with each I l0 class is a location called the interrupt vector. It cotains the address of the interrupt service pro celine All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the current) stach by the interrupt is removed and the stach points is set to point to a temporary stach used by the process handler. 1) Hardware stacks program counter, etc, 2) Hardware loads new program counter from internpt vector. 3) Assembly language procedure stres registes. 4) Assembly cangrage mocecure sets op new stack 5) C interrupt service uns (typically reads & lauttes input.) 6) schechle decides which process is to non next. 72 procedure returs to the assembly code.

loads up regrises, & memory map for o Assembly Congraze procedure starts up new winat process. the now- wrrent mocess & stanssit

running.

Modeling Multe programming **When mu*l*timogramming** is useel, the CPU utilization con be improveel.

Threads In traditional os, each process has an address space

and a single thread of control. In fact that is almost the definition

and as of a process. Wever the less, there are

• Nevertheless, there are frec*ve*ntly situations in which it is

frequently sou desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the odshared address space). Thread Usage The main reason for having threads is that in many applne

activities are going on aut g Threads is that in many applications multiple once. Some of these muy black from time to time. By decomposing such an app! sequenhal threa*c*h that run in *ana*s-pa*ra*llel, the programming in

. run in quasi-parallel, the programming model becomes simpler. It is the same argument for having mocesses. We can think about parallel processes, only now

nr abat parallel processes, only now with threachs we add in new element the ability for the parallel entities to share an address space and an themselves. A second cargument for threads is that they are lighter weight that processes, they are eamer (i.e. suster) to create and destroy than processes. When the number of threas needed changes dynamically and rapidly, this property is useful to have

ebul to have. A third reason for having threads is also a performance argument. Threads yield no performance gain

formance gain when all of them are cou bound, but when there is substantial computing and also substantial 110, having threads allows these activitres to overlap, thus speeeling up the application. Abo cheful on systems with multiple cous, where real parallelism is possible.

**07.**

Model

Threads Single-threaded process Finite-state machine

ocking rystem is interrupts

I characterstrus

Parallelism , blocking system calls No parallelism, blocking system cuells, sequentrul thread systen, pesumare P*ara*llelism*, non blocluing* syst*em cel*ls, interrupts The state of the computation must be explicitly rared and restored in a table even time the server switches from working on one regnert to another. In effect we are simultry threads & their stach the hard way. thall to progron,

that restored in a

A blocking function blocks the calling function. This means the caller does not do anything until the bloching function rotuns control to it. Synesroonised behar rour. Non-blochines functiow

and

finished. They simply call the function do not require the caller to wait until they are immediately carry on to the next instruction to exeute. This can be thought of an "asynchronous" interaction,

The Classic Thread model

sa The process model is based on two independent concepts: resource grouping and erewtion. Sormetimes it is useful *to* se*pa*ra*te brenn.*, this is where thre*ad*s c*o*me i

o cooarte them; this is were threads come in. A process thread has a program counter to keep track of which instruction to execute next. It has registers, which hold its variables. It has a stack, which contains the exewtion history, with one frome for each mocedre called but not yet returned from. Processes are used to croup resources together, threach are the entitis

chechuled for exention on the cro. Threads allon multiple exentions to take place in the same process environment, to a large deerree incependent of one another having multiple threads running in paradis one process is analogous to having multiple proces running in pealed in the cornwtor. In the formar threads share an adhess space and other

ther

reson

sources. In the latter case, processes share physical memy

doshs, printes & other resources of Multi threading when the CPU switches back and for the among the threacts, providing the illusion that the

I threads are running in parallel, albeit on a slower ceo than the real one. reaalfwrite rulle There is no protection between threads in the same procession wipe out a threads tach) as ou imposside and should not be necessary. Each trocadh will have a different execution history here it needs its own tach which contains fring for unreturned procedures which contain the procedures cocal variables and the return address to use when the povredio call his finished.

Threads

Thread

thread3

**LI**

Threadt's ! stadt

I

-

t Process Twend 3's stad.

[ Kernel

When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure, eg. thread\_create. It is not necessary for even possible) to specify any thing about the new thread's address space, since it automatically rung in the address space of the creating thread. The creating thread is usually returned a thread identifier that names the new thread when a thread has finished its work, it can exit by calling a library proceche say, thread exit(). It then vanishes and is no longer schedulable. Another common thread call is thread, greld. which allows a thread to voluntarily crive up the cou to let another thread ron, such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes. Implementing Threads in User space There are two main ways to implement a threads packaue in user space and in The choice is moderately controversial, and a hybrid implementation is also possible. The first method is to put the threads pachage entirely in userspace. The kernel knows nothing about them. As for as the kernes is concerned, it is managing ordinary single threached processes. An advantage is that a user-level threaels package ceen be implemented on an Os. that does not support threuds. Some os's fall in this catagory even now. All of these implemetur have the sae general structure. Threads run on top of a run-time sustem with

of a non-time system, which is a collection of procechures that manage threads (threant-createll etc...)

Process

Proces I T thread

un time nostem

L.Theats

Thread userspael

thread

~ Theels

•

Tread table

kernel

kernel

E

kerne

table.

***p*rocess** tuole

[user level threads package]

[threads package managed by the kernel]

When t*hreeees ar*e m*anag*eel in user sp*a*c*e, ea*ch process reeds its own private thread table to keep tr*ack* of the thre*a*ds in that proces*s. This* ***tabl*e is cina l**ogous *t*o the *k*ernel's *pro*cess table except that it keeps track only of the per- threal properties. threal's PC, regntes, sp etc... The thread tuble is managed by the run-time system. When a threact is moved to recely state or blocked steate, the information needed to restart it is stored in the thread tuble, eraethy the same wan the kernel stores in formation about processes in the process table. Thread switching is taste trapping into the kernel and is a strona argument in favour of user-level threacts packages. Local pro cechures such as thread-yield () and a thread schechler procedure are more efficient then kernel calls. No trap is needed, no context switch, memon cache need not be grashed so thread schechting is very User-level threus allow each mocess to have ce istomised algorithm. User-level threats hire issues with the implementation of blocking system calls suppose that a thread coolly reaus from the ten board before any keys are pressed Letting the thread actually make the rystem call is inacceptable, since this will stop all the threads Another moblem with user-level threach pachayes is that if a thread starts running, no other thread in that process will ever won unless the first thread voluntarily gives up the cpu. Within a single mocess, there the no clock interrupts, making it impossible to schedule mocesses nouvel-loobin fashin

thread

(takina turns). Unless a enters the non-time sesstem of its own free will, the schedules will never ge a

*T*he strongest argument against user-lev**el threa*d*s is th**at mouramm ers generally went threa*d*s

precisely in applications where the threads block often, like in a multithreaded web server. There **threuels are con**stantly, m*ak*ing system *calls.* One a treup has occurred to the kernel to carry out the system call, it is hardly any more work for the kend to switch threas if the old one is blocked, and having the kernel do this eliminates the need for constantly making select sys calo that check it reuel system calls one safe.

Implementing threads in the kernel

The kernel has a thread table that keeps track of all the threads in the system, when a thread wants to create a new threal on destroy an existing thread, it makes a kernel call, which then dues the creation a destruction by updating the kernel thread table. The Kernel's thread table holds each thread's registers, state and other information. The information is the same as with user-level threads, but now kept in the kernes instead of in user space (inside the runtime system). In addition, the kernel also maintains the traditional process table to keep track of processes. All calls that might block a thread are implemented as system calls at considerably crreater cost than a call to a non-time rystem procedure. When a thread blocks, the kemel, aut it cles wetion, can run either another thread from the same process (if one is ready) on a thread from a difteret process. With user-level threads, the runtime system keeps unning threads from its own process until the kernes takes the cou away from it (on there are no ready threats left to run). Due to the relatively greater cost of creating and destroying threals in the kernel, some systems take an environmentally correct a*pp*roach a*ne*l recycle their thre*a*ds. When a th reccel is destroyed, it is marked as not runnable, but its kernel clater structures are not otherwise affected. Later, when a new thread must be created an old thread is reactivated saving some overhead. Thread recycling is celso possible for user-level threads, but since the thread mange men overhead is much

maller, there is less incentive to do this. However some problems do still exist. For example, what happens when a multithreaded process forks? Does the new process have os many th reacts as the old One dich, or does it have just one? In many cases, the best choice depends on what the process is Planning to do next. If it is going to call exee to start a new program, probably one thread is the corect choice, but if it continues to exewte, reprodring all the threads is probably the right thing" Another issue is signals. Remember that signals are sent to processes, not to threads at least in the classical model. When a signal comes in which thread should handle it? Possibly threads could register their interest in certain signals, so when a signal came in it would be given to the thread that said it wants it. But what happens if two or more threaus registre for the same nig? Hy Gri*d* Implementations Various ways have been investigated to try to combine the advantages of user-level threads with Kernel-level threads. One way is to use kernel-level threads and then multiplex user-level threaus onto some or all of the Icernel threads. When this approach is used the programmer can determine how many kernel threads to use and how many user-level threa*d*s to multiplex on ea*ch o*ne. *T*his model gives the ultimute in flex ability. With this approach, the kernel is aware of only the learned-level threads and schedules those some of those threads may have multiple usr-level threads multiplerced on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threat in a process that was on an operating system without multithreaching capability. In this model each kernel-level thread has some set of user-level threas that take turns using it.

antiple user on a kernel

threats threend\_

ye.

kanel,

1

kernel

spere

te*ner*

Scheduler Activations While kennel threads are better than user-level threads in some key ways, they are also indispotably slower. As a con sequence, researchs hare looked for ways to improve the situation without giving up their good properties. Below we will descrite one such approach devised by Anderson et al called schechler activatrows. The goals of the scheduler cectivation work are to mimic the functionality of Kernel threads, but with the better performance and greater flexibility usually associated with threads

Packages implemented in user space. In particular, user threads should not have to make special kon blocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be Rossible to on other

threads within the same process if they are really Efficiency is achieved by avoiding unnecessary transitions between use and kernel space. If a thread blocks waiting for another thread to do. something for example, there is no reason to involve the kernes, thus ssaving the overhead of the Kernel-user transition. The user-space runtime system con block the synchronizing thread and

Schedule a new one by itself. When scheeller activations are used the kernel arrigus a certain nomber of virtual processes to each

process and lets the luser-space) ron-time system allocate threads to processes. This mechanism can also be used on multiprocessor where the virtual processurs may be real CPUs. The number of virtual process allocated to a mocess is initially one, but the process can ask for more and can aro retum rorocessos it no longer needs. The kemel can also take back virtual processus already allocated in order to cry

them to more needs processes. OS systems are written in e kanse it is powerful, efficient a predictable. Java is not pre*dic*table because it might wn out of storage at a critical moment and need to invoke the garbage collector to reclaim memory ofta most in opportune time. This cannot happen in C because there is no garbage collection inc.

Java supports monitors by adding the keyword synchronized to a method declaration, jura guarantee that once ang thread has started executing that method, no other thread will be allowed to start executing anis other synchronizeed method of that object. Synchronized methods in Sara chifter from classical monitors in an essential way: Sara does not have condition variables built in. Instead it oftes two procedures, wait and notify which when red inside synchronized methods, are not sobect to race conditions. Monitors are a programming language concent which programming language concept which

does not have. The compile nwst recognize them and a range for the mutex somehow. Another problem with monitors an*d* abo with semaphores is *that th*ey were designet for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By putting the sema phores in the shorted memory and p*r*otecting them with *TSL (T*est& Setteet) er XCHA instructions, we can avoid races, when we go to a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area network, these primitings become in applicable. The conchison is that semaphores are too low level and monitos are not sable except

for a few programming languages. Also, none of the primitives allow information exchange between machines. Something else is needed message passing Scheduling

In addition to picking the right process to run the scheduler also has to worry about making efficient use of the cou because process switching is expensive. To start with a suitch from user mode to kernel inode must ceur. Then the state of the current process must be saved, includz storing its reposters in the process table so that they can be reloaded later. In many systems, the memory map leg-memory reference its in the page table) most be saved as well, Next'a new process must be selected by naning the scheduling algorithm. After that, the mu must be reloaded with the memory map of the new process. Finally the new process must be started, In addition to all that, the process switch orally invalidateds the entire memory cache, forcing it to be dynamically reloaded from the main memory twice (upon entering the kernel and bearing it)

When to Schedule

Akey is related to scheduling is when to make A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed. Frost, when a new process is created a decrsron needs to be made whether to run the parent process or the child process. Since both process are in ready state it is a normal scheduling deerrron and can go either way that is the schedules can legitimately choose to run either the parent or the child next. In Pintos it depends on the priority. second a scheduling decision must be made when a praess exits. That process can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes. If no process is reader, a system- su*p*plied idle p*r*ocess is normally run. Third, when a process blocks on Ilo on a semaphore, or for some other reason another mouss has to be selected to run. sometimes the reason for Gloching may play a role in the choice. For example, if A is an important process and it is waiting for B to exit its critral region, letting B run next will allow it to exit its critical region and thus let A continue. The trouble, however, is that the scheduler generally does not have the necessary information to take *t*h*is depend*eney into acc*o*unt.

Fourth, when an 110 interrupt occurs a scheduling decision may be made. It the interrupt came from an Ilo device that has now completed its work, some process that was blocked waiting for the I/O may nou be ready to run. It is up to the scheduler to decide whether to

on the newly ready process, the process that was running at the time of the intempt, or some third process. If a hardware cloch provides periodic interrupts at soor 60 Hz or some other frequency, a scheduling decision can be made at each clock interrupt or at every koth cloch interrupt. Scheduling algorithms can be divided into two categories with respect to how they deal with cloch interrupts. After clock interrupt processing has been completed, the process that was running before the interrupt is resumed, unless a higher priority process was waiting for a now-satished time out. In contrast a preemtive schechling algorithm prehs a process and lets it run for a maximum of some fixed time. It it is still running at the end of the time interval, it is suspended and the

schedule, picks another process to run (if one is available). Doina preemtive scheduling regums having a cloch interrupt ocur at the end of the time interal to give control of the CPU bach to the scheduler. If no cloch is available, non preemptive scheduling is the only option.

Categories of S*chedul*ing Algorithms A s*chedul*ing algorithm for any systems should have 3 core *behav*iovis. Fairness:gring euch morens a sair share of the CPU. Policy enforcement - seeing that stated policy is carried out. Balance - keeping all parts of the system busy. Another general goal is too keep all parts of the system busy when possible. If the CPU and all the I/O devices are running all the time, more work gets done per second than if some components are idle.

First-come First-served

breemotive The simplest scheduling algorithm is non preemptive first come

first come frut-seres. With this

frut-semes. With.

algorithm, process ennest it. There is a single are

the CPU in the assigned

order they request it. There is a single queue of read

queue of really processes. Processes are not interrupted if they have on too long. When the running process blocks, the frost process on the queue is on next, when a

when a

bloched process becomes rea

bloched process becomes ready take a newly arrived ob it is out on the end of the queue. Easy to understand and easy to implement. Also fair, Disadvántare

This scheduling method is non preemptive that is, the process will run until it finishes. Because of this

scheduling, shoot

of the non preemplive

processes which are at the bach

queue have to wait for

This scheduling the scheduling

the finish.

Shortest job first

Another non-preemptive algorithm that assumes that run times are known in advance. Only optimal when all jobs are arailable simultaneously. Ready queue is treated as a priority queue based on smallest CPU time requirement. Priorities are assigned in ander inverse order of time needed for completion of the entire job. If equal time of completion, then FCFS is used for assigning priority. Arriving jobs inserted at proper position in queue, Dropatcher selects shortest job (14 in queue) and nas to completion. When multiple batch pots are sitting in a queue with the same priority, the schedules was the shortest job first. It cannot be implemented at the level of short term CPU scheduling Advantages: Minimises average waiting time, minimizes average turnaround time, provably optimal throughput is high. Dis advantages: In general connot be im*pl*ementedirequires future knowledge, in practice can't actually predict the length of next burst, can lead to unfairness or starvation

it may penalize processes with high service time requests. It the ready list is saturated, then processes with large service times tend to be left in the ready lost whith while small processes receve severe

In extreme case, where the system has little idle time, processes with large service times will never be Roesnt always minimize average turnaround time. Shortest remaining time next A preemptive version of shortest job frost is shortest remaining time next. With this algorithm, the schechler always chooses the process whose remaining un time is the shortest. Again here, the non time has to be known in advence, when a new pos arrives its total time is compared to the current process remaining time. If the new job needs less time to finish them the current process, the current process is suspencled and the new job startech Allous short jobs to get good service Round- Robin Scheduling One of the oldest, simplest, fairest and most widely used algorithes is the RRs. Each process is assigned a time interval, called a quantum during which it is allowed to run. It the process is still running a*t t*he end of the q*ua*n tumb, the CPU is pree*m*ted and given to *ano*ther p*r*ocess. I*f* t*he* in

process has blocked or finished before the quantum has elapsed, the croswitching is done when the process blocks. RR is easy to implement. All the scheduler needs to do is maintain a list of nnnable mocesses, when the process uses up its quantum, it is put on the end of the list The only interesting issue with RR is the length of the quantum. Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and lists, flushing and reloading the memory cache etc. If the quantum is set longer than the mean CPU burst, preemption will not happen very often. Instead, most processes will perform a bloching operation before the quantum runs *o*rt, causing a moress switch. Eliminating pree*mptio*n *imp*roves performance because poress ourtches then only happen when they are logically necessary when a process blocks and cannot continue). The conclusion: setting the quantum too short causes too many process Context switches and lowers (PU efficiency, but setting it too long may cause poor response to short Interactive requests. A quntum of around 20-50 msec is often a reasonable compromise Priority Scheduling RRS makes the implicit assumption that all processes are *equall*y imp*o*rtant. Each process is assigned a priority, and the rennable process with the highest priority is allowed to run. Is prevent high-priority processes from running in definitely, the scheduler may decreases the priority of the wrrently wnning moress at each clock tich. If this action causes its priority to drop below that of the next highest process, a process switch occurs. Or, each process may be assigned a maximum time quantum, that it is allowed to run. When this quant the next highest priority process is given a chance to run.

Hottey Scheduling

The tearic idea is to give processes lottery tickets for various system resources, such as cpu time whenen. a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. More important processes can be giren extra tickets to increase their odds of winning. In contrast to a priority schedules, where it is hard to state what having a priority of 40 actually means, here the nile is clear: a process holding a fraction of of the tickets will get about as fraction f of the resource in question. Lottery scheduling has several interesting properties

For example, if a new process shows up and is grented some tichets at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. Ls is highly responsive

Tickets can be exchanged between processes *T*hread scheduling A major d*iffer*ence between user-level thre*a*ds cunil kernel-level threads is *t*he performance, Domga thread switch with user-level threads takes a handful of machine instructions with icemed level threads it requires a full context switch, changing the memory map and invalidating the cache, which is several order of magnitude slower. On the other hand, with Icemet-level threads, hewing a thread

since the process as it cloes with user-level threads bloch in I/O does not suspend the entire kernel knows that switching from a thread in process it to a thread in process B is more expensive

and how the then running a second thread in process A Sare to having to change the memory map memory cache spoiled), it can take this information into account when making a decriron. For example given two threads that are otherwise equally important, with one of them belonging to the same process as a thread that just bloched and one belonging to a different process, preference could be given to the former.

OS in

ROM

Device Drivers in RO.

poznan

IRAM

Memory Management No Memory Abstraction

This model presented the programmer with physical memory and it would not be possible to non two programs in memory at the same time as they can overwrite each others values in physical memory. Howers even the model of memory being just physical memory, several options are possible.

user Models and chave the

This model is useel

*A* BIOS program / disadvantage that a bug in the

y some hand user program can wipe out the

user held comptes

OS in

operating system, possibly with

program

and embeded dronstrous results (gorbling the clist)

Systems.

osaam When a system is organized in this way, generally only one mocess at a time can be running however even with no memory abstraction, it is possible to run multiple programs at the same time. What the os has to do is save the entire contents of memory to a disk file, then bring in and on the next program. As long as there is only the program at a time in memory, there are ho conflicts. (swapping come ot. If two procpams are loaded into memory and one references the absolute physical memory of another the other program then it will coach. What we want is for each program to reference a private set of addresses local to it. A solution is to use static relocation where the second waled noorum is modified on the fly ruch that the address it was loaded at, say 16,384, the address constant is added to every program address chring the

local process. This works but it is not a geneal solution, holc Prog 2 slows down loading and requires extra information In all eventuble programs to indicate which words contain (relocatable) addresses and which do not

20+ 16384

34

A Memory Abstraction : *Addr*ess spaces

The notion of an address space

Two problems have to be solved to allow multiple applications to be in memory at the same time. without their inte fering with each other protection and relocation. We looked at a primitive solution to the former used on the IBM 360: label chunks of memory with a protection key.. and compare the key of the executing process to that of evey memory word tetched. Howers this approach by itself cloes not solve the latter problem, although it can be solved by relocating procpams as they are loaded but this is a slow and complicated solution. A better solution is to invent a new abstraction for memory: the address space. Just as the process concept creates a kind of abstract cpu to run

ner of abstract cpo to non programs, the address space

on creates a kind of abstract memory for programs to live in. An address space is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes special case where som shere, as it want to c.e. address 28 in one program means a different physical lantion than address 28 in auth mogram.

Base & Limit Registers

This simple solution uses a particularly simple version of dynamic relocation, what it coesis map each process address space onto a clifferent part of physical memory in a simple way. The CPU has two special hardware registers, usually called base and limit registers, when a process is on the base register is loaded with the physical address where its program begins in memory and the limit register is loaded with the length of the progrum. I.E. O 16,384, 16,384 32,768 (186)

Every time a process references memory, either to fetch an instruction or read or write a data worel, the CPU hardware automatically adds the tase value to the address generated by the process before senery the address out in the memon bus. simultaneously, it chechs if the address offered is equal to an greats them the value in the limit reenst, in which case a sow it is generated and the access is aborted

Swapping

If the physical memory of the computer is large enough to hold all the processes the schemes described so for will more or less do. Qut in practre, the total amount of RAM needed by all the

processo is often much more than can fit in memory. Two general approaches to dealing with menory overloul have been developed over the years. The simplest strategy, called swapping, coum of bringing in each process in its entirety winning it for a while, then putting it back on the disk. Iole processes are mostly stored on disk, so they do not take up any memory when they are not woning (although some of them wake up periodically to do their work, then go to sleep again). The other strategy, called virtual memory, allows programs torn even when they are partially in main memory. A point worth making concerns how much memory should be allocuted for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: the operating system alocates exactly what is needed, no more no less. It, however, processes' datar segments can grow, for example by dynanaly allocating memory from a heap arth a problem occurs when a

I ostach process tries to grow It a hole is adjacent to the process, it

- room process allowed to grow into the hole. can be allocated and the

for

.-datur I growth On the other hand is the process is aduent to another process, the

В-p*roty*ram growing process will either have to be moreel to a hole in menory large enough for it, or one or more processes will have to be

Astanh swapped out to create a large enough hole. If a process cannot grow in memory and the swap area on the drsk

A dah

growth is full, the processes will have to sospent until some spore

A-program is treed up Corit can be killed).

os

for

7?- room

Mengapa bertentang Virtual Memory while base and limit registers can be used to create the abstraction of address spaces, there is another problem that has to be solved: managing bloatware. While memory sizes are increassy rapidly, software sizes are increasing much faster. As a consequence of these developments, there is a need to un programs that are two large to fit into memory, and there is certainly a need to have systems that can support multiple programs running simultaneously, each of which sits in memory at which collectively exceed memory. Swapping is not an attractive option, since a typical SATA dich has a peach transfer rate of at most 100 MB/sec, which meus it takes at least 10 see to swap out a 1 GB procpam and another 10 see to swap in a 1-63 procjum. The basic idea behind virtual memory is that each program has its own address space, which is broken up thao chunus called pages. Each page is a contiguous range of addresses. These pages are mapped onto physical memory, but not all pages have to be in physical memory to nn the program. When the program references a part of its address space that is in physical memory the hardware performs the necessary mapping on the fly. When the program references a pert of its address space that is not in physical memory, the Os is alerted to go get the missing piece and re-exeate the instruction that failed. In a sense, VM is a genealrsatus of the base and limit register ideer.

Poging

When VM is useet, the virtual addresses do not go crrectly to the memory bus. Instead they go to a

mmu (Memory Mungemet Unit) that maps the virtual addresses into the physical memory addresses.

Virtual address

when the program tries to access address o

space

fer exam*p*le, using the instuction

GOK- 64K

mov REG O

SGK - 60k

IX - virtual

virtual address is sent to the mmo. The M MV sees

paze 52 k-56k

(0-4045)

that this virtual address salls in page

48K - S2k

which aunding to its mupping is page frame 2

(8192-12287). It thus transforms the address to 8192 44K-48k

and outputs address 8192 onto the bus. The memory HOK - 44k 36k-Lok

Phynal knows nothing at all about the MMO and just

32 K- 36k

memory sees a request for reading on writing addres

anoress 28 K-32 k

128k-32k 1192 which it honon cobites of offset in this 24K-28K

24k-28k

If a program references an unmapped addrelle 20k - 24K

2ok 24k

on like 32780 (which is byte 12 within Vittal

16k-20k

k-20k

page 8 starting at 32768) then the mmy notices that

12K-16k

12 -16K

k the page is unmapped and causes the CPU to trap xk-12k

8k-12k

to the os. Trap is called page ault. Evictron 4K - 8k

4k-8k

occurs and the needed page is brought into menog OK - дk

40k-4k

frame Page tables In a simple implementation, the mapping of virtual addresses onto physical addresses can be summarized as: the virtual address is solit into a virtual page number ( high-order bits) and an offset (low under bits). The virtual page number is used as an index into the page table to find the entry for that virtual page. From the nove table enty, the page grame number is found (if anyd. The page frame number is attached to the high-order end of the offset, replacing the virtual page number to form a physical address that can be sent to the memory.

hit

Structure of a page table entry

The referenced bit is set whenever

Page

a page is referenced, either

frame referred moulibal poteetich

for

number

reading or writing. Its rahe helps

the os to choose a page to evict present (1) labsent (0)

when a nage fault occurs. witel cachoy

e xeste bits disable

o keep track of page usage

e vrage

7 This

This bit is of valve when the ws decides to reclaim

no cache

is writtendo modified is - a page frame. If the page in it has been modified (1.e. for 110

I set to I Carrty bit)

dirty), it must be coritten back to disk. It it has not

been modified (lie. is clean') it can iust be a

I since the disk copy is still valid Translation Lookaside Buffers Let us now look at widely implemented schemes for speeding up nagng and for handling large virtual address spaces, starting with the former. The storting point as most optimization techniques is that the page table is in memory. The TLB is based on the observation that most programs tend to make a large number of references to a small number of pages, This only a small saction of the page table entries are heavily read the rest are barely seletal The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The TLR is usually inside the MMU and consists of a small number of entries (up to 64). Each entry containe information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (rlwlx permissions) and the physical page frame in which the page is located. Also has valid bit. How a This functions: when a virtual address is presented to the MMO for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (in parallel). It a valid match is found and the access does not violate the protection bits, the page frame is tahen directly from the ThB, without going to the page table, when the virtual page number is not in the TLB, the MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLS and replaces it with the page table entry just looked up. Page tables for large memories -how to deal with very large virtual adviers spaces, Multi-level page table A MLPT has a 32-bit virtual address that is partitioned into a 10-bit Pil & 10-bit PITZ hield and a 12-bit Offset field. The secret to the multilevel nage table method is to avoid keeping all the rage tables in memory all the time.

Inverted Page Tables A different solutron is neeeled for 64-bit naged virtual address spaces, Address space is now 2.64 bates, with 4-kB pages, we need a table with 20 entries. It each entry is 8 bytes the table is over 3OPB.One solution is the inverted page table. In this de

ye table In this design, there is one entry per page frame in real memory, rather than one entry por pages of virtual address space The entry keeps tracks of which process, virtual page is located in the page frame. Downside is that virtual to physical translation becomes uch harder. When process a references virtual page p. the hardware can no longer find the physical page by using p as an indes into the page table. Instead, it must search the entire inverted page table for an entry (no). Further more this search must be done on every memory reference, not just on page faults. This is not optimal so a 728 is used on a hit translation can happen just as fast as with reppler page tables On a TLB miss , however, the inverted rage table has to be searched in software. Best to use hashmas. lashed on virtual address.

**Page replacement algorithms** When a page fault occurs, the Os has to choose a page to evict to make room for the incoming page. If the page to be removed here's been mo*d*ified while in me*m*ory, it must be rewritten to *t*he disk to *b*ring the disk copy up to dete. If however the page has not been changed, the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the ppage being evicted. In all the *pa*ge re*pla*cement *algo*nthms to be st*ud*ie*d belo*w, a certain issue a*rr*ises i when a paye i*s to be* e*vi*cted

from m*emo*ry, *d*oes i*t ha*ve *to* be one o*f th*e fa*ulti*ng process *own p*ages, or can it *be a* p*a*ge *b*e*lo*nging to another process? In the former cese, we are effectively limiting each process to a fixed number of

pages, in the latter cose we are not The optimal page re*p*lacement algorithm **I marine e*uch* page is l*ab*e*l*ed with the numb**er of instructions t*h*at w*ill b*e exew teel *b*efore the*st paye* is first referenced, the optimal page replacement algorithm says that the page with the highest label should be removed. The only problem with this algoathm is that it is

m uth this alewithm is that it is unreal izable. At the time of the page facit the us has no way of knowing when each of the pages will be referenced nert. Useful as a benehnok ***The* not r**ecently used page rep*lac*ement a*l*gorithm 4 couche ap*pr*ox*i*mat*i*on o*f L*RU. E*a*ch p*a*ge has two status bits associated with it. Ris set whenever the page is refe*r*enced (re*ut/* written). Mis set when the page is modifie*d (* written to). The bits are :

contained in each pause table entry. These bits are updated on every memory reference, so are set og hared ware. Os resets *them. Th*e Ranel 4 *bi*ts c*an* be used to build a s*imp*le parering *algo*rithm as follows. When a process is started up, both page bits for all its pages are set to O by the os. Periodically on each clock int*er*r*upt*), the Rbot i**s cle*ared*, to *c*l*i*stingu**ish pages that have not *b*een referencecl recently from those **that have *b*een. Wh*e*n a *pag*e fault** occur*s*, *t*he os inspect*s all th****e pa*ges a*nd d*i*vid*es *them into 4*** categories Class 0: not referenced, not modified class 1 occurs when a class 3 page has its & bet Class 1: not referenced, modified

cleared by a clock interrupt. NRU algorithm removes a Class : referenced, not modified page at random from the lowest numbered non-empty Cluss 3: refere*nce*d, modified.

class. Gives *ut*egnute promene. Implit idea *th***at it**

a is better to remove an unreferenced moclifieel page than d

e

a *cl*e*an pa*sje in heer*y \**

First-In, First-Out Pag**e replacement algorithm The OS maintains a list o*f* all payes w*r*rently in m**emory, with the most recent *ar*rival at t**he t*ail and* the le*a*st recent *ar*rival at the heud. *O*n a p*age fau*lt, the page at the *head*** *is remov*e*d and the ne*w page added to the tail of the list. Rurely used, migh throw out important pages. second chance page replacement algorithm tas ***A* simple m*odificat*ion to FIFO that armo*ids* the problem of throwing out a he*a*vily use*d* pag**e is to ins*p*ect the R bit of the oldest page. It it is o, the page is both old and unused, so it is replaced immediately. It the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages and its load time is updated as *t*hough it had just a*rriv*ed in memory. Then the search continues. During evertion it the passe is *cir*t*y* (has been modified) it is written to disk, on just abandoned if it is clean. What second chance is looking for is an old page that has not been referenced in the most recent clock interval. It all the pages have been referencend, second chance degenerates into pure Fifo. precifically, imagine that all the pages in )

of the list, clearing the Re bit each have their R bits set. One by one, the os moves the pages to the end time it appends a page to the end of the list. Eventually, it comes back to page A, which now has its R but cleared. 14**4 this poin*t A* is evicte*d. Thus the a*lgorithm a*l*ways terminates. Pertanto** celeyo.inima Clock Page replacement algorithm Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is consta**ntly moving p**ages around on its list. A better *approac*h is to keep all the p*a*ge frames on a circlar list

page being in the

fault occurs, the form of a clock, The hand points to the oldest page, when a page pointed to by the hand is inspected. It its R bit is o, the page is evicted, the new page is inserted inte the clock in its plence, and the hand is advanced one position. If R is 1, it is cleared and the hand is

ndvanced to the next page. Process repeated until a page is found with R=0.

*R*ealistic.

ins a list of all pages wments Fault, the page cut the heaven

The heast Recently Used (LRO) **Raste repl*ac*eme**nt silyo nithm

A good approximation to the optimal algorithm is based on the observation that pages that have a been heavily used in the last few instructions will probably be heavily used again in the next few.

Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizuble algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linkeel list of all pages in memory, with the most recently used page at the front and the least recently used page cut the back. The diftiuitly is that the list must be upd*a*te*d* on every memory reference. Fin*d*ing a *pag*e in the list, *de*leting it, and t*h*en moving it to the front is a very time consuming operation, (even ith hardware ititurld be built!). Could have a 04 bit counter which is incremented after each instruction. Each page table entry has a field which contains that counter. On page fault, os examines all counters in page talle could have a hardware muhix of nen bits for a pages all o. On page reference k row koits set to 1 and column k bits set to O. At ang instant of time, the row whose binary value is - towent is the LRU. Although both of the previous LRU algorithms are (in principle) realizable, few. if any machines have the required hardware. Insted a solution that can be implemented in a sofware is needed - Not Frequently used (NFU). Aging works as follows: It requires a software counter associated with each page, initially o. At each clock interrupt, the os scens all the pages in memory. The counter is shifted right 1 bit before the R bit is added to the leftmost bit. when a page fault occurs, the page with the lowest counter is chosen for replacement. A page that has not been reference **of fo*r* a while will have more leading** ze*r*os. [2 difts withi LRU → *p*209]

Efurient algo that approx*i*mates *L*RU w*ell.* **Working set page replacem**ent algorithm

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, caurines the os to bring in the page containing the first instruction. Other paese faults for global variables and the stack.. usually follow auickly. After a while, the process has most of the pages it needs and settles down to non w*it*h relatively few page *fa*ults. *T*his strategy is **called *d*e*ma*nd pa**ening because paces are loaded only on demand, not in advance. The set of pages that a process is wrrently using is known as its working set. If the entire working set is in memory, the process will ***r*un without *caus*ing meine faults until it moves into anothe**r exeation *phas*e. Many p*ag*ina **systems try** to keep tra*c*k of each mouers' workin**g set and m*ak***e s*ur*e *that it i*s in memon, before letting the process run. Os has to keep track of which pages are in the washing set whene rage fault occurs, find a page not in the working set and evict it. Somewhat expensive to implenet *T*he *W*orking Set Clock (WC) pase *repla*cement algorithm *The* b*e*ernic working se*t algo*rithm is *c*umbersome, since the entire ***poen*e t*abl*e *ha*s to be sc*an****ned cat eac*h page fault until a suitab*le candidat*e is located. An improved algorithm, that is *based* on *the clo*ck amaith but also uses the working set information is called Wsclock. Due to its simplicity of implementation

*i*t is widely *us*e*d* in practice. *The* data struct*ur*e ne*ed*e*d* is a cirwlar list of . page frames, as in the clock alcorithm. Initially, this list is empty. When the first page is loaded, it is

o into the list to form a ring. Euch entry contains added to the list. As more pages are added, they go in the Time of last we field from the basic working set algorithm, as well

set algorithm, as well as the R&M bits. As with the clock algorithm, aut euch page fault the page pointed to be the hand is examined first. If R=1, the page has been used during the wrrent tick so it is not an ideal candidate to remove . R is set to o. rond advanced to the next page and the algorithm repeated for that page. If the page pointed to nas R=o. It the age is greater than I and the page is clean, it is not in the working set and a relid copy exists on the disk. The page frame is simply claimed and the new page out there. On the other hand, if the page is dirty, it cannot be claimed immediately since no valid copy is. present on disk. To avoid a process switch, the write to disk is scheduled, but the hand is aduenned

an old, clean page further and the algorithm continues with the next page. After all, there might be town the line that can be used immediately. In principle, all pages might be scheduled for disk TIM ano cercle around the clock. To reduce disk traffic, a limit might be set, allowing i

maximum of n nages to be written bash. Once this unit has been reached, no new writes are scheduled what happens if the hand comes all the way around to its starting point? There are acoses:

• At least I write has *b*e*en* sc*hec*hiled to

• Wo writes hav**e been s*c*he*d*uled**

In the first case, the hand just keeps moving, looking for a clean page. since one or mornites have been scheduled, eventually some write will complete and its page will be marked as

clean. The first clean page encountered is evicted. This page is not necessarily the first write scheduled because the disk driver may reorder writes in order to optimize disk performance. In the second case, all pages are in the working set, otherwise at least a write would have been **scheduled, La*c*hing** additional information, the rimplest *th*ing to *d*o is **cl*ai*m any *cl*ean pase ainel** se it. The location of a clean page could be kept trach of during the sweep. If no clean page *e*xists, then the w*rre*nt page is chosen as the victim and written *b*uh to *dr*ik. G*ood e Hi*cien*t* celeyo.

..

Design issues for paging Systems

**Local vs. global a*llo*cation p*o*l*ici*es**

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue astoricte**d, with this choice is ho**w memory shoul*d be all*outeel among **the competing nunnable proce**sses, Suppose pro*c*e**ss A gets a page fawit. Shoul*d* the** page rep*la*cement algorithm ty to find the least recently used page considering only the pages allocated to process for should it consider all the pages in memory? Local vergt u page replacement. Local algorithms effectively correspond to allocating every process a fix*ed* fr**action or t**he memory Globud algorithms dyna*mical*ly **al*locate pa*ge frames among the annu ble processes. *Thus t***he *nu*mber of page frames assigned to each process varies in time. In general, global algorithms work better, especially when the working set size can very over the life time of a process. If a local algorithm is used and the working set " grows, thrashing will result, even if there are plenty of free page frames. It the working set shrinks, local algorithms wast**e memory. If a glo*b*al algorithm is use*d, t*he system must continually decide**

process. The how many page frames to assign to each

tells when to

Page Fault Frequency algorithm increase or decrease a process page allocation but says nothing about which page to replace on a sault, Each process was a minimum number of pages so that all process sizes can exente and the number of pages is proportional to the process size, but the allocation has to be updated dynamically as the processes It is important to note that some pege replacement algorithms can work with either a local replacement policy or a global one. For **example, FIFO can re*p*l*er*ce the oldest** parje in all of mem*o*ry (g Go Goul) core the oldest page owned by the current process (local), similarly Leu or some approximation of it can replace the LRU page in all of memory or in the current process. On the other hand, for wither page replacement algorithms, only a local strategy makes sense. In particular, the working set and wsclock algorithms refer to some specific process and must be applied in that context. There really is no working set for the machine as a whole, and trying to use the union of all the working sets would lose the locality property and not work well. Load Control Even with the best page replacement algorithm and optimal global allocation of page frames to process, it can happen that the system thrashes. In fact, when ever the combined working sets of all processes exceed the capacity of memory thrashing can be expected. One symptom of this situation is that the PFF algorithm indicates that some processes need more memen but nopeesses need less memory. In this case there is no way to give more memory to those processes neeling it without hurting some other processes. The only real solution is to temporarily get rid of some process using swapping them to clisk and freeing their pages which they are holding. Swapping occurs until thorosty

Page Size Determining the best page size requires balancing several competing factors. As a result, there is no

noullantinum To start with, there are two factors that argue for a small pay size. A randomly chosen text, dut a on stack segment will not fill an integral number of pages. On average, half of the final

page will be empty. The extra space in that page is wasted. Another argument for a small page is that in general, a large page size will cause more unused program to be in memory than a small page size. On the other hand, small pages mean that programs. will need many pages hence a large page table. A 32-kB program needs only four 8-k13 pages, but 64 512 byte pages. Transfers to and from disk are generally a page at a time with most of the time being for **the seek and rotational delay, so that transferring a small page *tahes almo*s**t as *much ti*me as transferring a large page. It might take 644 Lomsec to load 64 512-byte reages, but only 4x12 insee to load 48-kB pages. The optimum page size is 2 ki. Shared pages

In Unix, after a fork system call, the parent and child are required to share both program text and data. In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set of pages. Thus no copying of pages is done at fork time. However, all the data pages are mapped into both processes as READ ONLY. As long as both processes just read their data without modifani oy it, this situation can continue. As soon as either process updates a memory world, the vision of the read-only protection causes a trap to the os. A copy is then made of the offending page so that each process now has its own private copy. Both copies are now set to READ-

WRITE, so subsequent writes to either copy moreed, without trapp*i*ng. This strategy means *that* those pages *th*at are never m*od*ified (in*clu*cling all *t*he

program pagesd need not be copied, Only the centar pages that are actually modified need to be copied This approach, called copy on write, improves performance by reducing copying, sharing can be done **at othe**r granularities theen inclividual parejes. I*f a pr*oepraim is started up twice, most *os'*s will **automatic**ally share all the text pages so that only one copy is in *m*emory. File Systems There are three essential *r*e*quir*ements for long-te*rm* information storage:

It must be possible to store a very large amount of information. 2) The information must survive the termination of the process using it. 3) Multiple time Eilen are berical units of information created by processes. They are stored on disks mest disks can be divided up into one or more partitions, with independent file systems on each partition, Sector o of the disk is called the MBR (Master Poot Recorel) and is used to boot the computer. The end of the MER contains the partitio**n table. This *tab*le gives the starting and en*d*ing a*dd***resses of each partition.

h artitio*n*s in the **t*ab*le is marke*d as ac*tive. *W*hen the *co*mputer is booted, th*e BI*OS *r*e*ad*s and** exentes the MRR. The first thing the MBR program does is locate the active partition reuel in its first bort called the boot block, and exewte it. The program in the boot bloch loads the os contained in that partition For uniformity, every partition starts with a tout block, even if it does not contain a bootable ***op*e*rati*ng sy**stem. Bes*id*e*s, it m***ight contain one** in the future. **Implementing Files: Contiguous Allocation** Contiguous disk space allocation has two significant advantages. First, it is simple to implement because keeping trach of where a file's blocks are is reduced to remembering two numbers: the disk awress of the first block and the number of blocks in the file. Given the number of the first bloch, the number of any other Hock can be found by a simple addition. Second, the read performance is excellent because the entire file can be read hom the disk in a single operation. Only one sech is needed ( to the first Hoch). After that no more seeks or rotational delays are needed, so data come in at the full brend width of the clisk. Thes ontiguous allocation is simple to implement and has high performance. Unfortunately, contiguous

ratim lama sanificant drowbach: over the course of time, the dist becomes fragmented.

allocation*,*

tome in the la*st Hoch*). Also it is sufficient on

the

ther hand, although reaning

Les vies disk bro

w

nber. Such a

data. Furthermo

Linked hist Allocation

phyrtual block The second method for storing files is to keep each one as a linked list of of each bloch is used as a pointer to the next one. The rest of the bloch is for

disk blocks. The first word

clarta. Unlike contiguous allocation, every disk bloch can be used in this method. Wospace is lost to clist fragmentation (except for internal fragmentation in the last block). Also it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there. On the other hand, although reading a file sequentially is straight forward, random access is extremely slow. To get to block n the os has to start at the begining and read the n-1 blocks prior to it, one at a time. Clearly doing so man reads will be pain fully slow. Also, the amount of data storage in a block is no longer a powercy two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less eficient because many programs read and write in blocks whose size is a power of 2, with the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to

- copying. Linked List Allocation Using a table in Memory Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each chik block and putting it in a table in memory. File Auses disk blocks 4, 7, 2, 10 and 12 in that order, and file is uses dist blocks 6,3, 11, 14 in that onder. Both chains are terminated by a special marter (-1) that is not a valid bloch number, such a table in memory is called a FAT (File Allocation Table) Using t***h*i's organization, the ent**ire Hock is *av***a*il*a*bl*e for data**. Furthermore, re*nd*om aceess is *much* easier. Although the chain must still be followed to find a given offset within the file, the chain is **entirely in memory, so it can be followed w*ith*out mak*ing* an**y *dis*k refe*r*ences, like *t*he p*r*evious method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is. The comer method is that the entire table must be in memory all the time to make it work with a 20048 dist and a 1 -KIB block size, the table needs 200 million entries, one for each of the 200 million dist blocks. Euch entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. This the table will take up 600 MO on 500 MP of main memory all the time, dependling on whether the system is optimized for space or time, Not practical, FAT idea does not seats well to terge disks. Physnal

**I-nodes** blood o

Our last method for keeping track of which blocks belong Z & Afile starts here

to which file is to associate with each file a data

struct*ur*e *called a*n e-no*d*e, *wh*ich lists th*e at*t*ri*b*ut*es **and** Jeo File starts here disk addresses of the file's blocks. Civen the i-node, it is

then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in memory table is that the i-node neeel only be in memory when the corresponding file is open, It each e-node ocopies n by tes anela maximum of k files may be open at once, the total memory occupied by the

array holding the i- nocles for the open files is only nk by tos File Attitutes

Only this much space need be reserved in actrance. This Awr of disch blocko

array is usually for smaller than the space occupied by *all* 17

the tile table described in the previous section. The reason -1 at

is simple. The table for holding the linked Gat of all disk **( 3 >**

blocks is proportional in rice to the disk itselt. In contrast

A*l*l4H

the i-node scheme requires an array in memory whose

size is proportional to the maximum number of tiles that

pointers

Voisk slock I

may be opened at once. One problem with a nodes is that it I containing habe each one has room for a fixed number of arst addresses

what happens when a file grows beyond this limit? One sootron Arst awrens not for aclata kloch out for the adress of one two blocks containing more disk bloch

Jo-NM shoorn 3 5 2 =

kunuted block

Adreds of block od

addresses