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 Cor supervisor mode). In this mode it has complete access to all the hard ware 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, these instructions that a tect contol of the machine or do I/0 ove for Gidden to user-mode programs, The user interface program, shell on GUL, is the lowest level of user-mode software,

Gare berdware and and allows the user to start other programs sweb

ser to start other programas s web browser). Os nous on provides the base for all other software. An important distinction between the operating system and normal Cuper mode ) softwere is that if a user does not like a particular mecil receler, she is free to get I write Chether one ; (she is not gree to white their own clock intempt handler, which is rent of the OS and is molected by bardware against attenpts by users to moc by

os The distinction is some times blurred in embedded on interpreted systems (such as sava-besed that use interpretation, not hardware to seperate the components, Everything wnning in Kernel mode is clearly part of the OS, but some programms running outside it are arguably also part of it, are closely associated with it like File Sytems & password changing, What is an OS?

Software which runs in temel mode. It basically bor forms two purelated functions: providing

umers, naturally) a clean abstract set of application programmers (and application programmers, naturally) a clean as resurses instead of the messer hard were ones and managing these hard were 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 hard were and present mograms (and their programmes with nice, clean, elegant, consistent, abstractions to work with instead. Os's real customers are the application programs (Lia 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 3 Gul I shell. In an alternative (bottom-up)

provide for an ovelonder and controlled allocation at the processes view, the job of the OS is to provide

the 106 of thes Os is to

for an excerley and contro memories, and I lo devices among the various prograns competing for them. In short, this view of the QS 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 conftretry requests from different programs and users, Resarces one sterreed in two ways > time & space. Computer Hardware review Program counter contains the memory address of the next instruction to be fetched. After

seram counter is no detect to point to its successor, the instruction has been fetched, the program counter is updated to point.

in memory. The stack contains one freeme Stack pointer points to the top of the current stuck

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

*Il*0 Devices I10 can be done in 3 ways. One 7 ( simplest) a ser program issues a system call, which the Kernel then translates into a procedure call to the appropriate driver. The driver then starts the I10 and sits in a tight loop continuously polling the device to see if it is done, when I10 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 distadventy, of teging up the CPU polling the device until it is finished. The second method is for the driver to start the device anel cash it to give een interrupt when it is finished. At that point the driver returns. The os then blocks the caller if needed and looks for other work to do. When the controller detects the end of the transfer, it generates an interrupt to signal completion. 3 ways use a DMA chip & Direct mening Access).

Os concepts - Processes In many oss, all the information about each process, other than the contents of its owns address space, is stored in an os table called the process table, which is an array for linked list) of structures, one for each process currently in existences System Cells Making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not. OS structure There are around six different Os structures including mono lithic systems, lengered systems, micro kernels Client-server systems, virtual machines and exo kenels. Monolithic Systems In this approach the entire OS runs as a single program in Kernel mode. The OS is written as a collection of moreclures, linked together into a single large executable binary program.

one. Having for senets of procedures Each mocedure in the system is free to call any other that can call each othere without restriction often leaves to an unwielelly and difficult to understand system

procedure, There is no information hiding, every procedure is visible to every other Basic structure of OS:

- A main program that invokes the requested service procedure

A set of gervice procedures that carry out the system walls.

• A set of utility procedures that help the service procedures. Layered Systems

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

Micro kernels

The basic idea behind the micro kenel design is to achieve high reliability, by splitting the OS up

the micro Kernel ruas in leemel mode and the into small, well-defined modules, only one of which rest on as relatively powerless andinery user processes. faster as a bug in the Kernel can Grily down the system instantly. About to lugs por 1000 lines of code.

Processes and threads

switching back and for the difference between a process and an

as a program, input, output cure

Process; an abstractions of a wnning program. Then support the ability to have (pseudo) concurrent operations even wh*e*n *th*ere is only ore CPU available. T*he*y tum a single CPU into multiple virtual CPUs. Processes

The Process M*o*del A process is just an instance of an execting program, including the current valves of the program counter reenisters and variables. Conceptsälly, each process has its own wirtual CPU. In reality of course the real CPU switches back and forth from process to process, out to understand the

parallo about a collection of mocesses running in (psevelod system, it is much easier to think

the CPU switches from mogram to program. This rapid than to try to keep track of how

č po switches rüpidly back chanel forth

called multi programmina switching back and forth is amarry the processes. The ditterence between a process and a program is sootle, but criticada The key idea is that a process is an activity of some kind. It has a program, input, output onnel astate. Ida mogram is running twice, it *co*unts as *t*wo processes. Process Creation There are four mincipal events that cause processes to be created:

System initialisation 2)

running process, Execution of a process creation system call by a

process 3) A user request to create a new 4) Initiation of a butch gob, Processes that stay in the background to handle some activity such as esmail, web pages, news,

one system call to create a printing and so on are called daemons. In Only there is only

exact clove of the calling process. After the forks new process: fork. This call creates an the two processes the parent and the child, leave the same memory image, the same environnet

executes execve on a similar strings, and the same open open files. Usually, the child process then

program. In both UNIX and windows, aggrem call to change its memory image and run a new

fter a process is created, the parent and child have their own distinct address spaces, If either process cleanese a word in its address space, the change is not visible to the other process,

the parent's, but there are definitely two In Unix, the child's initial address space is a copy of distinct address spaces involved; no writable memory is sherred. It is however possible for a

some of its creators other resources, such as open files. In newly created process to share

are different from the start, Windows, the parent's and child's address spaces

Process Termination A process terminates due to one of the following conditions. 1) Normal exit (voluntary) 2) Error exit (voluntary } Erogram bug) 3) Fatal exit ( involuntary) (by han eller). 4) killed by another process (involuntary)

In some systems, when a process terminates, either voluntarily or otherwise, all processes it creates are immediately killed as well. Neither Onix nom windows. werk this way, however,

*P*

Process Hrerarchies. In somme systems, when a process creates another, process, the parent process and child process con line to be associated in certain ways. The child process can it self create more processes, forming a process hierachry. In UNIX a process and allo dits children, together form a process group. Windows does not have a mocess herradwy. Processes in UNIX Cannot dis inhen't their children.

Process States

( Running)

Alth el mocess is an independent entity, with its commogram counter and internal state, processes often need to in teract with other processes. One process may generite some output theat another process uses as inout. 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.

Fraurition I scars when the external event for which a

**1**

process was waiting (such as the arrival of some input) happens. If no other mocess 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 it's turn comes.

Transitions between states.

The lowest level of the OS is the scheduler, with a variety D 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. core. The rest of the OS is nicely structured in process form. Few

reale systems are as nicely structured on this

ol 1l.... la-l

(Blocked)

( Ready

processes

ters

í skeduler Implementation of processes To implement the process model, the OS maintains a table can 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 blochead state so that it can be restarted later co if it back hever been s to onedo Associated with each I 10 class is a

contains the address of the interrupt service pro cedine location called the interrupt vector. It All interrupts start by saving the registers, often in the process table entry for the current process, Then the information pushed on to the current) stack by the interrupt is removed and the stack points is set to point to a temporary stack used by the process handler. D Hardware stacks program counter, etc, 2) Hardware loacks new mognum counter from interrupt vectos. 3) Assembly language procedure seves registers. A) Assembly congrage mo cecture sets up new stack she interrupt service wins (typically reads & Gen Hers input.) 6) schedules decides which process is to run hest. 72c procedure returs to the assembly code.

loads up registes, & memory map fure

the now" wrrent mocess &gtests it or Assembly Engrage procedive starts up new cunet process. <

running.

Modeling Multe programming **When mu*l*timogramming** is useel, the CPU utilization con be improveel. Threads In traditional OS, each process thas an address space and a 8

and a single thread of conhol. In fact that is almost the definition of a process. Wever the less, there are frequently sou

• Nevertheless, there are frec*ve*ntly situations in which it is desirable to have multiple threads of control in the same address space running in quasi-paralled, as though they were (almost) separate processes (except for the ad stared address space), Thread Usage The main reason for having threads is that in many cappure

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

rum in quasi-parallel, the programming model becomes simples. It is the same argument for having mocesses, we can think about parallel processes, only now

mr wont parallel processes, orrly, now with threads we add in new element: the ability for the parallel entities to share an address space and an themselves. A second caregement for threads is that they are lighter weight that the processes, they are carrer

of threas needed changes dyremially Core. faster) to create and destroy then processes. When the number

and to have. A third reason for having threads is also a per fermarie and rapidly, this property is useful to have argument. Threads yield no performance gain

formance gain when all of them are cou Gound, but when there is substantial computing and also substantial 110, having trends allows these activities to overlap, this specching up the application. Abo neful in systems with multiple cous, where real parallelism is possible.

**07.**

Model

Threads

Single-threaded process Finite-state machine

ocking system uts, interrupts I het restored in a

I Characterstres

calls Parallelism, blocking system

calls, sequestral thread system, no fumace Ne parallelism, Glocking system

P*ara*llelism*, non blocluing* syst*em cel*ls, interrupts The state of the computation must be explicitly saved and restored in a table even time the server switches from waring on the regnert to corretter, In ettent we are rimuktory, there areto & tless starts the hard way. Had to program,

A blocking function blocks the calling function. This means the caller does hat do anything

Gebeurour, Non-bloclines functions until the bloching, hunction etwas control to it. Synshensverse do not require the caller to wait until they the pirviskeet. They simply call the furiction and immediately carry on to the next instruction to exente. This can be thought of an "asynchronous" interaction, The Classic Threal models The mocess model is based on two independent concepts: resource grouping and execution,

- concerate them; this is were threads come in. A mocess thread how a program Sormetimes it is useful *to* se*pa*ra*te brenn.*, this is where thre*ad*s c*o*me i

te counter to keep track of which instruction to execute next. It has registers, which held

procesare variables. It has a stack, which contaires the exertion history, with one freeme for each collect but not yet returned fromm. Processes are used to group resources together; threats are the entitis checkled for exention on the ceo. Threads allow multiple exertions to take place in the same process environment, to a large deres inclependent of one another. Having multiple threatly running in parallelis one process is analogous to having multiple processes running in paralled in one cornuter. In the former

there a sources. In the latter case, processes shure physical memas threads otheme an access space and other resoure Lastes, printers & other resources, a Mv Hi threaching - when the CPO switches bach anel for the among the threaels, providing the illusion that the

realfasite van de werelle a threads are running in parallel, albeit on a slower cpu than the real one. There is ins mo tection between threads in the same process Cien wipe out a threadly stach) as or impossidie and @ shovid not be necessary. Each trocadd will have a different execution history where it needs its own stach which contains forwing for unretumad procedures which inte in the procedures Local verables and the return address to use when the procedere all his finished,

Threads

Thread?

thread 3

**LI**

Thread it's stacle

Kt Process + Tweled 3's stad.

E

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 libreng procedure, eg. thread\_create. It is not necessary for

runs in

thread's address space, since it auto mentically even possible) to specify anything about the new the address space of the creating thread. The creating thread is usually returned a thread identifier that names the news thread. When a thread has finished its work, it can exit by calling a library procedhe

thread call is thread, gield. say, thread.exit(). It then vanishes and is no longer schedulable. Another common which allows a thread to voluntarily give up the CPU to let another thread run, such a call is important

multiprogramming as there is with processes. because there is no clock interrupt to actually enforce

Implementing Threads in User space There are two main ways to implement a threads to aclame : in eser space and in The choice is moderately controversial, cand a hybrid implementation is also possible. The first method is to put the threads package entirely in user space. The Kernel knows nothing about them. As for as the kernes is concerned, it is managing ordinary, single threaded processes. An advantage is that a user-level three els package ceen be implemented on an Os that does not support threads. Some Os's fall in this category even now. All of these implemeta:

of a run-time system, which is a collectin have the same general structure. Threuels run on top of a run-time sustem with of procedures that manage threats Sthreemd.create() etc..)

Process Process

thread

Liseen's

Litgesels

pun time nostem

Thread

userspaeh

| Trend ttable

kernel

r

kernel

E

Kerme

table.

***p*rocess** tuole

[user level threads package]

[threads rachage 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- threaed properties :-> thregel's PC, registes, spete... The thread table is managed by the run- time system. When a threat is moved to recely state of

stored in the thread table, excetly the same was blocked state, the information needed to restart it is the kernel steres in formation about processes in the process table. Thread switching is seste trapping into the kernel and is a strona Ergument in favour of user-level threads packages. Local procedures such as thread yield () and a thread schechler procedure the nere efficient theen Kernel calls. No trap is needed, no context switch, memon cache need wat be grashed so thread scheckling is ver

customised algorithm. User-level threads have

from User-level threws allow each process to have a issues with the implementation of blocking system calls. So ppose that a threat tools reacts the ten board before any keys are pressed. Lebring the thread actually make the system call is inacceptable, since this will stop all the threads Another moblem with user= level thread packages is that if a thread starts wunning, no other thread in that

mocess, there the process will ever wn unless the first threed voluntarily gives up the cpu, within a single

feshin (taking turns). Unless e no clock interrupts, making it impossible to schedule mrocesses nouvel-besoin

thread' enters the wn-time sesstem of its own free will, the schedules will never get 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 Glock 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 kernel to switch threads in the old one is blocked, and having the kernel do this eliminates the need for constantly making select sys calls that check it real system calls one safe.

Implementing threads in the Kernel

Theo Kernel hees a threeed table that keeps track of all the threads in the system, when a threat wants to create a new threat or destroy can existing thread, it makes to learned call, which then dus the creation an destruction by updating the Kernel thread table. The Kernel's thread table Loles each thread's registers, state, and other in formation. The information is the same as with user-level threacs, but now kept in the kernes instead of in user space Cinside the unatime system). In addition, the kernel also main tains the traditional process table to keep track of processes. All calls that might Hocks a thread are implemented as system calls, at considerably creater cost than a

procedure. When a thread call to a run-time system

crocks, the kernel, cut it clescretion, can run either another threach from the scene process Citome is ready) on a thread from a different process. With user-level threads, the run . time system keeps running threads from its own process until the Kernel takes the CPU away from it can there care se reccely threats left to run). Due to the relatively greater cost of creating and destroying threevels 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 data structures are rest otherwise affected. Later, when

old thread is reactivated, saving some overheads Thread a new thread must be created, an recycling is celso possible foro user-level threaels, but since the thread mange mest overhead is much

maller, there is less incentive to do this. However some problemes de still exist. Fer erample, whead happens when a multi three elech process Barko? Does the news process brave as many threarly as the old one dich, or does it have just one? In menin cases, the best choice depends on what the process is planning to do next. If it is going to calleree to start a new program, probably one thread is a the corect choice, but if it continues to execute, reprodreing all the threads is probably the right thing Another issue is signals. Remember that signals are sent to processos, best 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 threat that said it wants it. But what happens if two or more thread's registre for the same rig? Hy Gri*d* Implementations Various ways have been investigated to try to combine the advantages of user-level threads with Kernel-level threeds. One way is to use kernel-level threads and then multiplex weer level threas onto some or all of the Kernel 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 ultimate in flex ability. With this approach, the Kernel is aware of only the Kernel-level threads and schedules those. Some of those threads may have multiple usne level threads multiplexed on top of them. These user-level threads are created, destroyed, canel scheduled just like user level threads in a process that was on an operating system without multithreading capability. In this model, each Kernel-level thread has some set of user-level threads that take terms using it.

multiple user treeks on a kernel threemad,

ye.

te*ner*

Takemells kernel

spere

Scheduler Activations While kemel threads are better than user-level threads in some key ways, they are also indisputably slower. As a come sequence, researchs have looked for ways to improve the situation without giving up their good properties. Below we will describe one such approach devised by Anderson et al called

work are to mimic the functionality of schechulen echivatrows. The goals of the scheduler cectivation Kernel threads, but with the better performance and greater flexibility usually associated with threach Packages implemented in user space. In particular, user threads Sterld not have to make special kon blocking system calls on ceek in advance if it is safe to make certain system cells. Nevertheless, when a threads Glerks on a system cell or on a page fault, it should be possible to run other threads with in the same mocess if they are recicls. Efficiency is achieved borg avoiding unnecessary transitions between user accel kernel space. If a thread Glocks waiting for another threach to do: something, for example, there is no reason to involve the kernes, thes saving the overhead of the Kernel-user transition. The user-space runtime system can block the synchronizing thread canel Schedule a new one by itself. When scheeller activations are used the kemel assigns a certain number of virtual processors to each process and lets the lusern space) ron-time system cellocate threads to processors. This mechanism can also be used on multiprocessing where the virtual processors may be real CPUs. The number of virtual process

and can abu retum processos allocated to a process is initially one, but the process can aok of om meore it no longer needs. The kemel čam abo tame back wirtual processus already allocated in order to organ

them to here needs processes. OS systems are written in e kanse it is powerful, efficient a predictable. Java is not pre*dic*table because it might win out of storage at a critical moment and need to invoke the garbage collector to reclaim memory sta most in opportune time. This cannot happen in C because there is no garbage collection in c. Java supports monitors. By adding the keyword synchronized to a method declaration, jove guarantes that once ang thread this started executing that method, to ather thread will be allowed to start executing canis other synchronized method of the object. Synchronized methods in Sava clitter from classical monitors in an essentral way: Javer does not have condition variables Guiltin. Instead it offers two procedures, wait and notify which the cred inside son chronized the thorts, are not subject to rove conditions. Monitors are a programming language concept which

programming language coment which C 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 seme proves in the shared memory and protecting them with TSL (Test & Setters) on xchu instructive, we can avoid roves, when we go to a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area netwish, these primitings become in applicable. The conchnion is that sema plores are too low level and monitors are not usable ercent for a few programming languages. Also, home of the primitives allow information exchange between machines. Something else is needed > Message passing. scheduling

In addition to picking the right prices to run the schediter also has to worry about making efficient use of the CPU because process switching is expensive. To start with a switch porn user made to kernel mode must occur. Then the state of the current process must be saved, includiya storing its registers me the process table so that they can be reloaded later. In many systems, the memory map leg, memory references bits in the page table) must be saved as well, Next á new process must be selected by runing the scheduling algorithm. After that, the MMV must be

must be stenter, In addition reloaded with the memory map of the new process. Finalling the new process to all that, the process switch usedlin invalidated as the entire memory cache, forcing it to be dynamically

entering the Kernel and learning it). reloaded from the main memory twice lupon

When to Schedule A key iste 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. First, when a new process is created, a decision needs to be made whether to run the parent process or the child process. Since both process are in reades state, it is a normal scheduling decision and can go either way that is the scheduler can legitimately, choose to run either the parent on the child next. In Pintos it depends on the priority.

second, a scheduling decision must be merde 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 I10, on a semaphore, or for some other reason, another mouss has to be selected to run, sometimes the reason for Glocking may play a role in the choice. For example, if A is an important process and it is waiting for B to exit its critical region, letting B run next will allow it to exit its critical region and this 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 I10 device that has now completed its work, some process that was blocked waiting for the I/0 may not be ready to run. It is up to the scheduler to decide whether to run the newly ready mrocess, the process that was running at the time of the interupt, or some third process. If a hardware clock provides periodice interrupts at 50 or 60 Hz or some other frequency, a scheduling decision can be more at each clock interrupt or at every kith clock interrupte Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. After clock interrupt processing has been completed, the process that was nnning before the interrupt is resumed, unless a higher priority, process was waiting for a now- satrihed time out. In contrast, a preemtive rehechtiva algorithm pichis o process and lets it run for a maximum of some fixed tince. If 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 requins having a clock interrupt occur at the end of the time internal to give control of the CPU bach to the scheduler. If no clock is available, non preemptive the dating 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 fair share of the CPU. Polity enforcement - seeing that stated policy is carried out. Balance - keeping all parts of the system Gusy. 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

greemotive first come font-sere. With this algorithm, process The simplest scheduling algorithm is non preemptive first come. Frat semes, with

mennest it. There is a single queue of really processes. Processes are assigned the CPU in the order they request it. There is a single queue of reach are not interrupted if they have un too long. When the running process blocks, the first process.

when a blocked process becomes ready, take a nemity arrived on the queue is won

got, next, when a blonec process becomes rec it is out in 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 hon me emp live scheduling, short processes which are at the bach of the quere have to wait free ima mrocess at the front to finish.

Shortest job first

Another non-preemptive algorithm that assumes that run times are known in advance. Only optimal when all pots are available simultaneously. Ready queue is treated as a priority queue based on smallest CPU time requirement. Priorities are assigned in order 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 pob (14 in quene) and was to completion. When multiple batch jobs are sitting in a queue with the same prority, 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, approvably 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 in fairness 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 cost which while small processes receive some In extreme case, where the system has little idle time, processes with large service times will never be Roesmet always minimize average turnaround time. Shortest remaining time next

A preemptive version of shortest job first is shortest remaining time next, with this algorithm, the schechler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance, when a new pas arrives, its total time is compared to the current processo remaining time. If the new job needs less time to finish them

Allows stunt pots the current process, the current process is suspended cance the new job startech to get good service, Round- Robin Scheduling One of the oldest, simplest, fairest and most widely used algorithis is the RRS. Each process is assigned a time interval, called a quantuma, during which it is allowed to run, It the process is still

in 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* process has blocked or finished before the quantum has elapsed, the CPU switching is clone when the process blocks. RR is easy to implement. All the scheduler needs to do is maintain a wist of runnable 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 memong cache etc. If the quantum is set longer then the mean CPU burst, preemption will not

before the quantum happen very often. Instead, most processes will perform a bloching operation

because runs *o*rt, causing a moress switch. Eliminating pree*mptio*n *imp*roves performance mocess suitches then only happen when they are logically necessary s when a process blouts and cannot continue). The conclusion: setting the quantum too short conses too many process luntext switches and towers CPU 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 rinnable process with the highest miority is allowed to run. In prevent high monity processes from running in definitely, the scheduler may decreases the priority of the wrrently running mocess at each clock tick. If this action causes its priority to

process may be drop below that of the next highest process, a process switch occurs. Or, each 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.

Hottery Scheduling

The basic 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 tichet is chosen at random, and the process holding that ticket gets the resource. More importent mocesses can be given extra tickets, to increase their odds of winning. In contrast to a miority schedules, where it is hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction f of the tickets will get about to fraction f of the resource in question. Lottery scheduling has several interesting properties For example, if a new process shows up and is granted 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 icerned hered threads it requires a full context switch, changing the memory map and invalidating the cache, which is several orders of magnitudes slower. On the other hand, with Kernel-level threads, having a thread block on 110 does not suspend the entire process as it cloes with user-level threads sime the Kernel knows that switching from a thread in process it to a thread in process B is more expenme

the then running a second thread in process A (dve to having to change the memory map and her mening cache spoiled), it can take this in formation into account when making a decriron. For example given two threeids that are otherwise equally important, with one of them beloriging to the same process as a thread that just blocked and one beloneng to a different process, preference could be given to the former,

OS in

ROM

Drivers in RO.

I program

| RAM ,

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 valves in playsical memory. Mower even the model of memory being just physical memory, several options are possible.

*A* BIOS This model is useet

Device 7 Models in de have the user

ing some hand program | disadvantage that a bug in the

user held commitas user program can wipe out the operating system, possibly with

program

and embeded OS in

draustrous results (garbling 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

are and run the next program. As long as there is only one mornam at a time in memory, there no conflicts. (swapping conce of). If two proprams are loaded into memory, and one references the itsolute physical memury of another the other program then it will crash. what we want is for Each program to references a private set of addresses local to it. A solution is to use static relocation where the second loaded nooram is modified on the fles, such that the address it was toaded at, say 16,384, the address constant is added to every program address during the

L a Prog 2 load, process. This works but it is not a general solution,

information slows down loading and requires extra in all exew tuble programs to indicate which words

20+ 16384 untuina (velocatable ) addresses and which do not

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 out the same time, without their inte fering with each other : protection and relocation. We looked at a mimitive tolution to the former used on the IBM 360: label chunks of memory with a protection key. and compare the key of the executaing process to that of every memory word fetched. Howert, this approach by it self does not solve the latter problem, although it can be solved by

and complicated solution. relocating programs as they are loaded, but this is a slow A better solution is to invent a new abstraction for memory: the address space. Just cas

of abstract CPU to run on the process concept creates a kind

mer og abstract cpo to run programs, the address space created as 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 Especial case where seem there, as it want to t.e. address 28 in one program means a different physical location then address 28 in anth mogram.

Base & Limit Registers This simple solution uses a particularly simple version of dynamic relocation, what it does is map each process' address species on to a different part of physical memory in a simple way. The CPU has two special hard were registers, usually called base and limit registers, when a process is ron, the base register is loaded with the physical celebress where its program begins in memory and the limit register is loaded with the length of the progrum, I.E. 0316,384, 16,384 7 32,765 (181) Every time a proceso references memory, either to fetch an instruction or read on write a data wered, the CPU hardware automatically adds the base valve to the address generated by the process before senely the address out in the memor Gus. Simultaneously, it checks if the address offereed is equal to or greater them the valve in the limit regnte, in which case a sewit is generated auch the access is aberted,

Swapping,

If the physical memory of the computer is large enough to hold all the processes, the schemes

of RAM needed by all the described so for will more or less do. But in practme, the total amount processes is often much more than can fit in memory. Two general approaches to dealing with memory over local have been developed over the years. The simplest strategy, called swapping, comm

hile, then putting it back on of bringing in each process in its entirety, wnning it for a the disk. I alle processes are mostly stored on disk, so they do not take up any memory when they are not running Calthough some of them were up periodically to do their work, then go to sleep again). The other strategy, called virtual memory, allows programs to run even when the

much memory should be u are partially in main memory. A point worth making concerns how allocated for a process when it is created or swapped in. If processes are created with a fixed

alocates exactly what size that never changes, then the allocation is simple: the operating system is needed, no more no less. It, however, processes dates segments can grow, for example by dynamely

occurs when a

Lo stach , allocating memory from a neap, as on a problem

Groom for mocess tries to grow. If a hole is adjacent to the process, it

- dute , growth can be allocated and the process allowed to grow into the hole.

В-p*roty*ram On the other hand, if the process is aciment to another process, the growing process will either have to be moved to a hole in memory

Astart large enough for it, or one or more processes will have to be

process

I Broom in swapped out to create a large enough hole. If a

A data i Melai |

gram cannot grow in memory and the swap area on the drak

LA- prognan is fuld, the processes will have to suspened until somente a space is breed up (or it can be killed).

Mengapa bertentang

Virtual Memory while beese 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 increasny

consequence of these developments, there rapidly, software sizes are increasing much faster. As a is a need to un programs that are too large to fit into memory, and there is certainty a need Po 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, smie a typical SATA dock has a reah transfer rate of at most 100 MB/sec, which means it takes at least 10 see to swap out a 1- GB propam and another 10 see to swap in a 1-GB program. The basic idea behind virtual memory is that each program has its own address spaces, which is Grohen up no chunus calleet pages. Each range is a contiguers range of addresses. These pages are mapped on to physical memory, but not all races have to be in physical memory to run the

that is in physical memory program. when the program references a part of its address space

the fly. When the program references a pert the hard were performs the necessary mapping on 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 generalisations of the base and limit register ideer.

ideer

Paging

when VM is used, the virtual addresses do not go directly to the memory bus. Instead they go to a MMV (Memory Mangement Unit) that maps, the virtual addresses unto the physical memory addresses,

when the program tries to access address o Virtual address

fer exam*p*le, using the instuction space

mov REG O GOK- 64K SGK-60K

virtual address o is sent to the MMU. The MMU sees Tx I B- virtual paze

that this virtual address falls in page 0 (0-4045) 52 k-56k

which acending to its mapping is page frame 2 48K-52K

(8192-12287). It thus transforms the address to $192 446-48K

anel outputs address 8192 onto the bus. The memory HOK-44K

physical knows nothing at all about the MMU and just 36K-LOK

memory Lees a request for reading on writing advies 32 K- 36k

aporess 28 K-32 k

1256-324 7192 which it lamang cobytes of offred in this 24K-28K

24k-28K

If a program references an unmapped add reite

201-241 206- 24K

them like 32780 (which is lyte 12 within Virtual

Ek-20k 16 K-206

page 8 tentry at 3276\*) then the MMV notices that

12 -16K 12K-16k

k the page is unmapped and causes the CPU to trap

8K-1216 xk-12k

to the Os. Trap is called page fault. Eviction

AK-8K 4K-SK

occurs that the needed page is brought into mencing OK - дk

40k-4k

page

frame Page tables In a simple implementation, the mapping of virtual addresses on to physical addresses can be summarized as: the virtual address is split into a virtual page number ( high- ander bits) and an oftset Clow-urde bits). The virtual, mage number is used as an index into the page table to find the entry for that virtual page. From the nese table enty, the page frame number is jouned (it any do 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 tables entry

The referenced but is set whenever

a page is referenced, either for

Page frame referenced modi bell protection

reading on writing. Its valve helps number

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

. when a page fault occurs. writel Cachry.

e xeste bits disable!

ver. keep track of rage usage 7

e usage 1. This Git is of valve when the Os decides to reclaim

This

a page frame. If the page in it has been modified live. is written de modified is no cache

dirty), it must be critten back to click. If it has not for I10

i det to I Carpty bit)

been modified Clie, is clean lit can just be a

Since the disk copy is still valid.

Translation Lookeside Buffers bet us now look at widely implemented schemes for speeding up nagmy and for handling Large virtual address spaces, starting with the former. The starting point of most optimization

on the observation that techniques is that the page table is in memory. The TLB is based most programs tend to make a large number of references to a small number of pages,

for This only a small fraction of the pages tables entries are heavily read; the rest are barely seletat The solution that has been devised is to equip computers with a small hardware device mapping virtual addresses to physical addresses without going through the page table. The TLR IS

small number of entries Cup to 64). Each entry containe usually inside the MMU and consists of a information about one page, including the virtual page number, a bit that is set when the range is

the modified, the protection code crlwlx permissions) and the physical page frame in which page is located. Also has valid bit. How a TLB functions; when a virtual address is presented to the MMU 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 paralel). It a valid match is found and the access does not violate the protection Gits, the page frame is tuhen directly from the THB, without going to the page table, when the virtual page number is not in the TLB, the MMU

evicts one of the entries from detects the miss and does an ordinary page table lookup. It then the TLS and replaces it with the pages tables entry just looked up. Page tables for large memories how to deal with very larger virtual adviers spaces, Multi-level page table A MLPT has a 32-bit virtual address that is partitioned into a 10-bit Plt of 10-bit PTZ field anel a 12-bit Offset field. The secret to the multilevel nage table method is to avoid keeping on the peeges tables in memory all the time.

Inverted Page Tables A different solution is needed for 64-bit nageed virtual acebress spaces, Address grace is now 204 lates, with 4-kB pages, we need a table with 23 entries. If each ening is 8 og tes the

ge tave. In this design, there is one table is over 30 PB. One solution is the inverted page table. In this de entry per penge frame in real memory, rather than one entry per pages of virtual address grace The entry keeps tracks of which process, virtual page is located in the page frame.. Downside is that virtual to physical translation becomes much harder. when process in references virtual page p, the hard were can no longer firect the physical page by using p as an inder into the page table. Instead, it must search the entire inverted page table for an entry (n, 0). Furthern more this search must be done on every memory reference, not just on page faults. This is not ontimal so a 728 is used. On a hit translation can happen just as fast as with regulero page tubes On a TLB miss however, the inverted page tuble has to be searched in software. Best to use hash mwe. lasted 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 date. If however the prege 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 page 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 case, we are effectively limiting each process to a fixed mumber of pages in the latter case 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 paas replacement algorithm says that the page with the highest label should be retroved. The only problem with this algorithm is that it is

M ut this adeo rithm is that it is unreal izable. At the time of the page feast the OS has no way of knowing when each of the pages will be referenced next. Useful ces as Genehmende ***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 page table entry. These bits are upcluteed 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, Goth page bits for all its pages are set to O by the os. Periodically con 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 1 occurs when a Class 0: not referenced, not modified

class 3 paese has its e árt

cleared by a clock interrupt. NRU algorithm removes as Class 2: not referenced, modified

page at random from the loweste numberedd non-empty Class 2: referenced, not modified

class. Gives *ut*egnute promene. Implit idea *th***at it** Cluss 3: refere*nce*d, modified.

a san is better to remove an unreferenced modlified 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 rage replacement algorithm das ***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* ***p*age is both old and unused**, so it is repl*ac*ed imme*d*iately. *It t*he 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 it it is clean, what second chance is looking for is an old page that has not been referenced in the most recent clock interval. If all the pages have been referencend, secossel chance degenerates into pure FIFO. Specifically, imagine that all the pages 49) leave their R Gits set. One by one, the Os moves the pages to the end of the list, cleaning the B bit each time it appends 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 in the form of a clock, The hand points to the oldest page, when a page fault occurs, the page being pointed to buy the hand is inspected. If its Ŕ bit is O, the page is evicted, the new page is inserted inte the clock in its place, and the hand is advanced one position. If R is 1, it is cleared and the hand is

advanced 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 thest pages that have. 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 sunuests a realizable 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 at the back. The difficulty 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 ish here were dititorld be built!). Could leve a 04 bit counter which is incremented after each instruction. Each page tube entre les a field which contains that counter. On page fault, Os examines all counters in page talle

row Kibits could have a hardware medix ef nen bits for a pages all 0. On page reference k set to 1 and column k bits set to O. At ang instant of time, the row whose binary valve is lowest is the LRU. Although both of the previous LRU algorithms are (in principle) realizable, few,

a if any, machines have the required hardware. Instead a sowtion that can be implemented in sofware is needed - Not Frequently used (NEW). Aging works as follows: It requires. a software wunter associated with each page, initially o. At each clock interrupt, the OS seems all the pages in memory. The counter is shifted right I bit before the R bit is added to the leftnost bit. when a

for replacement. A page that has page fault occurs, the page with the lowest counter is chosen 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 ties to fetch the first instruction, it gets a page fault, caurines the Os to brine, in the page containing the first instruction. Other page faults for global variables and the stack usually follow auickly. After a while, the process nees 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

currently rocesses are loaded only on demand, not in advance. The set of pages that a process is 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 working set, whenen rage fault occurs, fired a page not in the working set and evict it. Somewhat expensive to implement *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 worleiner sett 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 aterorithm. Initially, this list is empty. When the first page is loaded, it is

to into the list to form a ring. Each entry contains added to the list. As more pages are added, they go in

set algorithm, as well as the R&M bits. As with the Time of last wie field from the basic working set algorithm, as wel the clock algorithm, at each page fault the prese pointed to buy the hand is examined first. If est, the page has been used during the current tick to it is not an ideal candidate to remove . R is set to rand advanced to the next page and the algorithm repeated for that page. It to the page pointed to nas R=0. If the age is greater than t and the page is clean, it is not in the working set and a valid 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.

scheduled, but the hand is advanned present in disk, to avoid a process switch, the write to disk is and the algorithm continues with the next page. After all, there might be an old, dean page further town the line that can be used immediately. In principle, all nages might be scheduled for disk

Tin mo ano cirle around the clock. To reduce disk traffic, a limit might be set, allowing a

ma*ximu*m of n n*ag*es to be w**ritten bush. O*n*ce t*his lim*it h*a*s** bee*n reach***eel, *n*o *ne****w c*orites are

scheduled, what happens if the hand comes all the way around to its starting point? There are seases:

• 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, cooking for a clean page. Since one or mor mites have been scheduled, eventually some write will complete and its page will be marked ces 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 ander to optimize disk performance. In: the second case, all pages are in the working set, otherwise at least I 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** use 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 try to find the least recently used page considering only the pages allocated to process for should it consider all the pages in memory? Local verged 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

is used and the western set' set size cam very over the life time of a process. If a local algorithm grows, thrashing will result, even if there are plenty of free page frames. If the werking set shrinks, local algorithms wast**e memory. If a glo*b*al algorithm is use*d, t*he system must continually decide** how many page frames to assign to each process. The Page Fault Frequency algorithm tells when to increase on decrease a process page allocation but says nothing about which page to replace on a fault, Each process was a minimum number of pages so that all process sizes can exente ancel the number of pages is propertional to the processi size, but the allocation has to be updated dynamically as the processes It is important to note that some prege replacement algorithmes 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 Lev on some approximation of it can replace the LRU pace in all of memung on in the current process. On the other hand, for other page replacement algorithms, only a lored strategy makes sense. In particular, the working set and wis Clock 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 units 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 thrasties. 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 theat the BFF algorithm indicates that some processes need were memen but no processes nee less memung. In this case there is the way to give more memory to those processes meeting it without hurting some other processes. The only real solution is to temporarily get rid of some presses uring swapping them to cisk and freeing their pages which they are holding. Swapping occurs until thresby

Page Size Determining the best page size requires Galancing several competing factors. As a result, there is no B allanti dom To start with, there are two factors that argue for a small page size. A randomly chosen text, data on stack segment will not fill an integral number of pages. On average, half of the final paese 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 mure unused program to Ge in memory than a small page size. On the other hand, small puejes mean that programs will need mera pages, hence a large page table. A 32-kB program necels only four 8-k is peeges, 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 64x com sec to load 64 512-byte penges, but only 4k!2 mdee to load 26 8 - KB pages. The optimum page size is 4 kB. Shereed pages In Unux, 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 Goth processes as READ ONLY. AS Long as both processes just read their data, without modifesting it, this situations can continue. As soon as either process up cletes a memory word, the vision of the read-only protection causes de trap to the Os. A copy is then made of the offending rage so that each process now has its own private copy. Both copies are now set to REVAD-WEITE, so subsequent writes to either cupy moceed. without trapp*i*ng. This strategy means *that* those pages *th*at are never m*od*ified (in*clu*cling all *t*he program preges) need not be copied, Only the data, pages that are actually modified need to be copied, This approachs, 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 su*r*vive the te*rm*in*a*tion of the process whing it 3) Multiple time

on disks mest disks Eilen eve toerical units of information created org processes. Then the stored **can be d*ivid*e*d* up into *o*ne or more *p*artitions, with independent file syste*m*s *on eac*h partition, S*e*ctor *o*** of the disk is called the MERC Master Best Record) 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**

resel in its first exentes the MRR. The first thing the MBR program does is locate the active partition Work called the boot block, and exewte it. The program in the boot black loads the os contained in that partition. For uniformity, every partition starts with a book 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 Keeming track of where a file's blocks are is reduced to remembering two numbers: the disk address of the First block and the number of blocks in the file. Given the number of the first block, the number of any other **Hoch un be found by a simple a*d*ition. Seco**nd, the reed performance is excellent because *th*e entire file can be read from the disk in a single operation. Only one seek is needed ( to the first Hock). After that is more seeks or rotational delays are needed, so data come in at the full Grensel width of the clask. Thes on tiguous allocation is simple to implement and was high performance. Unfortunately, contiguous

fragmented. sătim laman sanificant drawbach: over the course of time, the disk becomes

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

play steal block The second method for storing files is to keep each one as a linked list of disk blocks. The first word of each bloch is used as a pointer to the next one. The rest of the block is for data . Unlike contiguous allocation, every disk block can be used in this method. No space is lost to clist freegmentation (except for internal fragmentation in the last black). 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 in, the OS las to start at the begining and reset the n-1 blocks prior to it, one at a time. Clearly doing so many 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, reeds of the full block size require acquiring and concatenating information from two cost 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 click block and putting it in a table in memory. File It uses disk glocks 4,7,2, 10 and 12 in that order, and file 13 uses disk blocks 6,3,11,14 in that order. Both crains are terminated by a special

FAT (File Allocation Table) marker (-1) that is not a valid block number. Such a table in the memory is called a 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*

offset within the file, the chain is easier. Although the chain must still be followed to find a given **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

(the starting bloch number) and still method, it is sufficient for the directory entry to keep a single integer be able to locate all the blocks, no matter how large the file is. The same method is that the entire table must be in memory all the time to make it work, with a 2004B

entries, one for each of the 200 million dist and a 1-KB block size, the table needs 200 million dist blocks. Each entry was to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 600 MO ir 500 MP of main memory all the time, depeneling on whether the system is optimized for space on time. Not practical, FAT idea does not seats well to terge disks. Phyginal

**I-nodes** book o

Our last method for keeping track of which blocks belony

to which file is to associate with each file a data 7

A file starts het

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

then possible to find all the blocks of the file. The Gia advantage of this scheme over linked files using in in memory table is that the s-node mesel only be in memory when the wresponding file is open. It each z-node occupies in bytes undla maximum of k files may be open at once, the total menury occupied by the

array holding the 2-modes for the open files is only nk bg tog File Attitutes

Only this moch space need be reserveet in advance. This Awr oddick blocko

array is usua**lly for smaller tha**n the *space oc*cu*pi*ed by -

It

the tile table described in the previous section. The reason - 12%

i's simple. The table for holding the linked Gist of all disk **( 3 >**

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

the i-mode scheme requires an array in memory whose A*l*l4H

of files that size is proportional to the maximum number

may be opened at once. One problem with an modes is that it painters

| containing the ab each one has room for a fixed number of arsle a donessen

B . what happens when a file grows beyond this limit? One solutrons ANSE Quer, hat for a data block out for the address of ane Hwo blocks containing more disk block

O-NML DI?

kunused block

Adress or better than

ypisk block

I may be a

Gedresses