Roteiro Sistemas Operacionais – Prova 1

Capítulos 1 à 17

Capítulo 4 - Processes

1. O que é um processo?

Basicamente um processo pode ser definido como um programa em execução . Qunado não está em execução um programa é apenas um apanhado de bytes descansando no Disco rígido. Quando são executado (através de um process API create) eles se tornam um processo. Uma CPU pode estar executando milhares de processos ao "mesmo tempo" através da virtualização da CPU e time sharing.

2. Quais os estados de um processo?

Basicamente os estados de um processo são Running , Ready e Blocked . e Necessário também citar que além desses três estados básicos processos também podem ter um estado inicial e um estado final.

3. É possível determinar em que ordem processos executam em um SO? Essa ordem pode mudar de execução em execução, memso se os mesmos processos estejam executando todas as vezes?

Não , como demostrado no capítulo 5 , muitas vezes o comportamento do CPU scheduler se mostra imprevísivel. Mas é importante citar que caso os processos sejam executados de certa maneira (utilizando wait() , como também demostrado no capítulo 5) , pode-se saber a ordem de execuçÃo, mas de maneira geral é impossível se prever isso.

4. Quais os dados devemos armazenar para poder gerenciar processos? Determine um exemplo de estrutura com esses dados

O sistema operacional deve manter , por exemplo , uma lista de processos para mapear que processos estão prontos (Ready) , aqueles que estão atualmente sendo executados e que estão bloqueados. Na sessão 4.5 do livro podem ser visualizados de froma geral os dados que são armazenados. Estão entre eles PID, Endereço de memória , Tamanho da memória do processo , "Pai"do processo e entre outros dados.

Capítulo 5 – Process API

1. Quais as três chamadas principais da API de um sistema UNIX são utilizadas para gestão de processos? Como cada uma delas funciona?

As três principais chamadas de um sistema UNIX são Fork () , utilizado para basicamente criar um clone de um processo em execução , com as mesmas informações de contexto , com a única diferença sendo o retorno da função fork() , sendo o PID do filho no pai e 0 no filho , Wait () , utilizado para esperar que certa execução se encerre , basicamente ele faz o que seu nome indica e Exec() , utilizado para substituis o processo

atual totalmente pelo processo desejado , como não há resquícios do processo que chamou exedc() , caso ele tenha sucesso , essa função nunca retorna.

2. Como funciona o fluxo de um programa que crie um processo? Como diferenciar se estamos no processo pai ou no processo filho? Como se dá o fluxo de cada um deles? Dica: a questão de fluxo tem relação com o estudado no capítulo anterior.

Basicamente o programa e chamado para execução e tem em seu código uma função Fork(), ele carrega o código e qualquer dado estático, que cria um processo. No Filho o fork retorna 0 , e no pai o fork retona PID do filho . Basicamente o Fluxo deles é bem semelhante , O código é carregado de um hard disk para a memória RAM , que gera o processo pai , que possui Memória e variáveis próprias. A partir dai é gerado o process da função MAIN , a partir da execução da função ao encontrar o comando fork(), que gera o processo filho , esse processo entretanto , não começa a ser executado em main , é quase como se ele tivesse chamado seu próprio fork. A Child não é uma cópia exata da pai , ela possui seus próprios endereços de espaço , memória própria e o retorno obtido da função Fork é diferente também. Em geral não podemos saber qual processo será concluído antes , a não ser que tenhamos um comando wait()

Capítulo 6 – Mechanism Limited Direct execution

1. Em que consiste o Limited Direct execution Protocol? Quais as questões que em que ele atua?

O limited Direct Protocol é um mecaninsmo criado para resolver a questão de time sharing inerente a virtualização da CPU , basicamente ele atua proporcionando a execução direta de programa na CPU . Para evitar falhas de segurança , o Limited Direct Execution protocol introduz o User mode , modo em que os códigos são executado que apresenta diversas restrições para o que se pode fazer e caso o programa necessite de alguma operação privilegiada ele pode efetuar uma system call , que permite que o SO libere certas partes de dados ou funcionalidades exclusivas do modo kernel . O SO necessita de uma sequência especial de trap instructions , que se autorizadas aumentam o acesso do programa para administrador , qunado finalizada a system call o SO retorna para o programa solicitante e baixa seu nível de privilgio através de uma return-from-trap function. Em geral , para recuperar o controle , o sistema operacional terá três opções , aguardar a execução da aplicação , aguardar ela tentar algo illegar , que devolve imediatamente o controle ao SO ou aguardar o timer interrupt que ativará o interrupt handler , timer responsável por devolver o controle das operações par ao SO dado certo período de tempo. Dessa maneira conlui-se também que o protocolo é adotado para fazer com que a virtualização da CPU ocorra da maneira mais eficiente possível em questão de velocidade , mas com seguraça e estabilidade.

2. Quais as maneiras de se mudar o contexto, ou seja, o processo que está em execução?

As maneiras de se mudar o processo que está em execução são através de conclusão do processo que retorno o controle ao SO através de Trap (software) , através da tentativa

de algo ilegal por parte do programa que está sendo executado ou finalmente , através do Timer interrupt , que através de um timer pré definido chama o interrupt handler que retona o controle para o SO através de instruções pré definidas. Essas três maneiras definem como se interrompe o atual processo para voltar para o SO , após isso a, através do scheduler o SO decide o que rodar em seguida e esse processo se repete infinitamente.

3. Como salvar e restaurar contexto? Quais dados são importantes?

O scheduler é responsável pela decisão de continuar no processo que estava sendo executado ou migrar para outro . Se a decisão é de mudar , o SO executa uma série de instruções de baixo nível que é referida como context switch. Tudo o que o SO tem que fazer é salvar alguns valores de registro 3 para o programa rodando atualmente e carregar os mesmos valores para o código que será executado em seguida. Para salvar os dados do programa que deixará de ser executado , o SO executa instruções assembly para salvar registros , PC e o Kernel Stack Pointer do programa e então restorá os mesmos dados do programa que irá ser executado , no caso da stack elas são apenas invertidas , de modo que o kernel lança uma chamada para alterar o código do antigo processo pelo novo . Quando o SO executa as instruções return-from-trap o context switch foi completdo.

Capítulo 7 – CPU Scheduling

1. Como funciona o algoritmo FIFO? Quais as vantagens e desvantagens?

É o mais básico dos algortmos de escalonamente , basicamente o primeiro processo a entrar na fila de processos será o primeiro a ser executado. É extremamente simples e fácil de implementar , porém quando se fala de Turnaround time se tem um sério problema com relação ao processamento de processos grandes , visto que processos menores podem ser “queued” por processos grandes , é o chamado convoy effect. Tal efeito aumento muito o turnaround time , tornando o FIFO ineficiente.

2. Como funciona o algoritmo Shortest Job First (SJB)? Quais as vantagens e desvantagens?

É uma evolução do FIFO onde o próximo processo a ser exceutado será o mais curto processo. Evita o convoy effect para a chegada de três processos simultaneos , mas caso um processo grande seja executado , poderemos ter o convoy effect . Além disso , caso tenhamos uma chegada constante de processos curtos podemos ter uma situação em que processos grandes demorem mutio para serem executados.

3. Como funciona o algoritmo Shortest Time-to-Completion First (STCF)? Quais as vantagens e desvantagens?

Política de escalonamento que adiciona uma condição em que toda vez que um processo novo entra na fila , o esclaonador determina qual dos processos da fila , incluindo o que está sendo executado , tem o menor tempo para ser completado, e o executa. Tal política resolve muitos dos problemas dos dois algoritmos anteriores e seria extremamente eficiente caso soubessemos o tempo para conclusão dos processos que chegam ao

escalonador . Quando se analisa esse algoritmo de um ponto de vista de tempo de resposta , pode-se ter um comportamento muito ineficiente , visto que os processos podem demorar muito para serem escalonados pela primeiro vez (Os processos com shortest time são completados até termos um novo escalonamento).

4. Como funciona o algoritmo Round Robin? Quais as vantagens e desvantagens?

Desenvolvido para solucionar a problemática relacionada ao tempo de resposta discorrido na responta anterior . Ao invés de rodar os processos até sua conclusão , o escalonador round-robin roda os processos por uma partição de tempo chamada “time slice” e então avança para o próximo processo na fila . O tempo do time slice impacta diretamente no tempo de resposta , quanto menor , menor o tempo de resposta . Contudo , ao mesmo tempo que o algoritmo de Roun Robin é ótimo para tempo de resposta , o turnaround time é péssimo , visto que os processos vão ser executados por uma porção de tempo pré definida e podem demorar muito até serem concluidos.

5. O que a ocorrência de operações de I/O introduz no modelo de escalonamento?

Quando um processo solicita um I/O seu status fica como bloqueado até a conclousão do I/O , dessa maneira , por questões de otimização , o escalonador tem de tomar duas decisões , uma quando o processo entra em I/O e outra quando o processo retorna do I/O. Em uma abordagem do escalonador STCF temos que o tempo para conclusão de um processo será dividido dado o número de I/Os que ele irá performar. De forma que que caso seja subdividido em processos de 5ms , esse será o tempo analisado pelo escalonador , e durante o I/O , outros processos serão executados para não termos uma IDLE CPU.

Capítulo 8 – The Multi-Level-Feedback-Queue

1. Como funciona o algoritmo MLFQ?

O MLFQ possui diversas filas , onde cada uma tem uma prioridade diferente e processos em filas com maior prioridade são escolhidos para serem executados, caso existam mais de um processo em filas de alta prioridade , normalmente algoritmos com RR são utilizados. Quando um processo entra no sistema ela será alocado na fila de maior prioridade , o algoritmo não sabe se o processo será curto , por isso recebe a maior prioridade. Para evitar que processos de maior prioridade sofram com “starvation” e permitir que processos possam se tornar interativos , dado um tempo S todos os processos serão movidos para a fila de maior prioridade. Quando um processo usa todo o seu slot de tempo , independente de quantas vezes ele deixou o CPU para efetuar um I/O ou algo do genero , ele tem sua prioridade reduzida. Dessa maneira , temos o termo Multi-Level justificado por termos diversos niveis de prioridade e o termo Feedback justificado por alterarmos a prioridade baseada no histórico do processo.

Capítulo 9 – Scheduling Proportional Share

1. Como funciona o escalonamento por loteria?

Baseado no contexto de Tickets , abstração utilizada para representar a porcentagem de CPU que um processo deva receber. O algoritmo atinge “fairness” probabilisticamente rodando uma lotéria constantemente. Dado um sistema de 100 tickets , um processo A recebe 75 tickets e um processo B recebe 25 tickets , sendo o processo A 0 – 74 e o processo B 75 – 99 , uma lotéria consistiria na geração de um número aleatório entre 0 e 99 para decidir qual processo será executado. Note que para um lotéria não é possivel saber qual processo será executado , mas quanto mais se executa a lotéria , se chega mais próximo da proporção inicial (probabilisticamente) . Contudo o ecalonamento por lotéria simples não é determinístico , dessa maneira , para se garantir a proporção , adicionamos o conceito de STRIDE , número obtido através da divisão de um número “grande” pelo número de tickets de cada processo. Assim , a cada vez que o processo for sorteado , soma-se esse número a um contador (pass value) para ter uma noção de de seu processo global. Esse valor será usado para garantir as proporções e tornar o escalonador determinístico.

2. Como funciona o escalonador do Linux, o Completely Fair Scheduler (CFS)?

É uma implementação altamente eficiente e escalável de um escalonador completamente justo , utilizando pouquissimo tempo para tomar decisões. O CFS utiliza uma técnica de contagem conhecida com VRUNTIME , quando uma decisão de escalonamento deve ser tomada o CFS escolhe o processo com menor vruntime para ser executado. Para determinar o quanto os processos devem ser executados (loteria) , em quesito de tempo , o CFS utiliza o SCHED_LATENCY que tem um valor típico de 48ms , e divide esse valor pelo número de processos para determinar o time slice. Para não se ter um time slice muito pequeno e um grande número de trocas de contexto utiliza-se o parametro MIN_GRANULARITY , para definir o valor mínimo do time slice , tem seu valor típico de 6ms. O CFS tem um timer que é disparado regularmente , ou seja , o OS só toma as decisões de escalonamento qunado o timer é disparado , onde ele determina se um processo chegou ao seu fim ou deve continuar sendo executado.

É importante ressaltar que o CFS adota Niceness (Nice level of a process) para indicar a priridade de um processo . Dado um niceness do processo , ele tem seu Vruntime alterado para dar a prioridade especifica. Quanto a organização da ‘fila’ de processos , o escalonador , por questão de otimização e agilização , o CFS organiza os processos sendo executados e executados em uma Red-Black-Tree. Quando um processo retorna após um longo tempo , seu VRUNTIME e setado para o menor da árvore , para evitar que este monopolize a CPU.

Capítulo 10 – Multiprocessor Scheduling

1. O que são os conceitos de localidade temporar e espacial?

São conceitos que baseam o funcionamento dos Caches. A ídeia pro trás de localidade temporal é que quando uma informação (Dado) é acessada , é provável que ela seja acessada novamente em um futuro próximo , como por exemplo variaveis e instruções sendo acessadas repetidamente em um loop. Já a ídeia pro trás da localidade espacial é que se um programa acessa dados em uma determinada localidade , é bem provável que as informações proximas de x também sejam acessadas em um futuro próximo , quando

um programa analisa um array , por exemplo. A partir desses conceitos , e o fato de que esses tipos de localidades são muito recorrentes em programas , os sistemas de hardware podem fazer um estimativa bem precida dos dados à armazenar na memória cache.

2. Qual o problema introduzido na questão de escalonamento em multiprocessadores quando pensamos em sincronização?

Com relação a sincronização , o principal problemas que enfrentamos é relacionado ao fato de multiplos CPUs acessando (e alterando) dados compartilhados e estruturas compartilhadas , de forma que é necessária a implementação de um mecanismo de limitação de acesso a esses recursos , como primitivas de exclusão mutua ,como LOCKS para garantir que as informações acessadas estão corretas. A adoção de locks resolve vem o problemas de concorrência , mas gera um problema de escalabilidade devido ao fato de , conforme o número de CPU ,. O acesso a dados compartilhados fica mais lento.

3. Como funciona o Single Queue Multiprocessos Scheduler (SQMS)? Quais os problemas apresentados por esse algoritmo?

Funciona reaproveitando os escalonadores de singleprocessor , basicamente aloca todos os processos em um única fila , sua vantagem é a simplicidade de implementação. Contudo , o SQMS não é muito escalável , e para garantir seu funcionamento correto é necessário adicionar alguma forma de “Locking” no código. Contudo , a adição de “Lock” reduz a performance do sistema , particularmente conforme o número de CPUs sobe. O segundo problema séria a afinidade de Cache, visto que temos apenas uma fila para multiprocessadores , é mais eficiente que um processo seja executado em um processador que tem suas informações armazenadas em sua memória cache , dessa maneira é necessário que o SQMS implemente um mecanismo de cache affinity , para executar , se possível , no mesmo CPU que estava anteriormente. O algoritmo é simples de implementar , dando a existência dos algoritmos singleprocessor , mas claramente não se mostra escalável e não preserva de afinidade de cache de maneira exemplar.

4. Como funciona o Multi Queue Multiprocessos Scheduler (MQMS)? Quais os problemas apresentados por esse algoritmo?

Ao contrário do SQMS , o MQMS não possui apenas uma fila global para alocar os processos , mas cada processador tem uma fila própria de processos que seguirá um algoritmo específico de escalonamento (Como RR , por exemplo) , em tese os processos são escalonados independentemente , evitando os problemas citados no SQMS . O MQMS é bem mas escalável e proporciona uma afinidade de cache inata , contudo ele pode apresentar um novo problema , o load imbalance . O load imbalance consiste em basicamente uma distruibuição incorreta de tempo da CPU , quando , por exemplo , uma CPU está se dedicando a um processo apenas , e outra CPU está executando multiplos processos . Para resolver esse problemas , devemos efetuar o que é chamado de Migration , migrando , quando necessário , um processo de uma CPU para outra , atingindo assim um verdadeiro equilibrio. Também podemos citar a técnica conhecida como work stealing , onde uma fila que está com a carga de trabalho “espia”

outra fila e checa sua carga de trabalho , podendo “roubar” um ou mais processos dessa fila caso esteja cheia. É importante ressaltar que caso essa checagem se torne muito frequente , perderemos muito desempenho , encontrar o equilibrio , como em todos os tópicos de sistemas operacionais , é necessário.

Capítulo 13 - The Abstraction: Address Spaces

1. O que é espaço de endereçamento?

O espaço de endereçamento é uma abstração da memória física que contém toda a memória física do programa. O código do programa, a stack e a heap estão dispostos dentro desse espaço de endereçamento. Contudo , é importante ressaltar que tal espaço é a abstração que o sistema operacional está fornecendo para o programa em execução , de maneira que o sistema está virtualizando sua memória , e o endereço físico de memória que está sendo utilizado , não é de fato , aquele do espaço de endereçamento , mas o endereço físico não está disponível para o programa devido a necessidade de transparência.

2. Em um modelo simples que não contempla multiprogramação, qual a direção de crescimento da heap? E da stack? Qual a razão dessas direções?

Em um modelo simples que não contempla multiprogramação o Heap cresce no sentido positivo , a partir do fim do código do programa e a stack cresce no sentido negativo a partir do fim do espaço de endereçamento. Tal disposição ocorre para que ambos possam crescer de forma mais correta. Dividir o espaço de ambos em dois acarretária em que ambos teriam apenas metade do espaço disponível.

3. Quais as preocupações em termos de transparência, eficiência e proteção devemos ter em mente quando se trata de gestão de memória?

As preocupações que se deve ter , quando falamos de gestão de memória são , transparência , de modo que o sistema operacional deve virtualizar a memôria de forma que seja invisível para o programa , eficiência , tanto em espaço , não utilizando muitas memória para armazenar as estruturas necessárias para a virtualização , quanto em tempo , não fazendo com que os programas rodem de maneiro lenta devido a virtualização , proteção , de forma que o sistema deve proteger um processo e suas informações de outros processo , de forma que quando um programa está rodando , não deve de maneira alguma ter acesso as informações de qualquer outro programa ou do próprio OS.

Capítulo 14 - Interlude: Memory API

1.Entender as questões envolvidas nas chamadas de malloc(), free(), principalmente o tratamento de retorno, a não alocação de memória antes de utilizá-la, as consequências de não desalocar memória alocada dinamicamente

A função malloc() tem um funcionamento bem simples , basicamente ela serve para alocar memória dinâmicamente , na HEAP . Toma como referencia o tamanho do

espaço à ser alocado e , caso suceda , retorna um ponteiro para esse local em que a memória foi alocada , caso falhe , retona NULL . A função free é utilizada para liberar esse espaço alocado dinamicamente pela função malloc , diferente de malloc , que toma como argumento o tamanho à ser alocado e retorna um ponteiro , a função free toma como argumento apenas o ponteiro e o tamanho da memória alocada é reponsabilidade da bilblioteca de alocação de memória. Caso o usuário não aloque memória para rotinas que necessitem de alocação de memória , como strcopy , teremos como resultado uma segmentation fault. Para casos de não alocar memória suficiente , podemos ter um buffer overflow , o que retorna um erro imprevisivel , visto que não sabemos o que está gravado nos endereços próximos ao endereço alocado. Caso a memória seja alocada mas não seja preenchida teremos a leitura de um valor desconhecido. Com relação a não desalocar a memória alocada dinamicamente , teremos um erro conhecido como memory leak. Em programas e eventos que rodam durante longos períodos , a não liberação da memória alocada dinâmicamente incorre no fato de que o heap é todo ocupado , podendo fazer com que o programa fique sem memória disponível , obrigando um reset. Desalocar memórias antes de utiliza-lá completamente e liberar uma memória duas vezes podem resultar em crashs.

Capítulo 15 - Mechanism: Address Translation

1. Em que consiste realocação dinâmica? Como ela funciona? Quais são as funcionalidades de hardware necessárias para que este método possa ser utilizado?

A realocação dinâmica consiste em uma técnica de conversão de espaço de endereçamento em endereço físico de memória baseado na utilização de hardware. Também referido como Base and bounds funciona com o auxílio de dois registradores para cada CPU (base e bound). Tais registradores permitem alocar o espaço de endereçamento em qualquer lugar da memória física , enquando garante que o programa acessará apenas seu espaço de endereçamento. Seu funcionamento se dá alocando o programa como se fosse armazenado no endereço zero , e quando o programa começa a ser executado , o SO decide onde , na memória principal , o programa deve ser carregado e os registradores são alterados para esses valores. Após essa alocação , toda referência a memória será referenciada somando o endereço virtual ao registrador base (indica o começo da memória física) . Devido ao fato da realocação do endereço ocorrer em runtime ,essa técnica é chamada de realocação dinâmica. Com relação ao registrado Bound , ele é utilizado para definir o fim da memória do programa, de forma que caso o processo gere um endereço virtual superior ao registrador , a CPU iniciará uma exceção e o processo será encerrado. Tais registradores são armazenados no MMU (Memory Management Unit) parte do processador que ajuda na tradução do endereço de memória.

2. Como se dá o fluxo da Limited Direct Execution Protocol (Dynamic Relocation)?

O fluxo do Limited Direct Execution protocol de dá da seguinte maneira , o sistema operacional aloca um entrada para o processo a na tabela de processos , aloca memória

para o processo e seta os registradores de base e bound e return-from-trap para a. Os registradores de a são então restaurados e o modo alterado para user mode , passa-se para o PC inicial de A. O processo roda e dá fetch nas instruções. O hardware faz a tradução do endereço e performa esse fetch , o programa então executa as instruções , o hardware então vai chegar a existência de um load/store explicito , chegar a legalidade do endereço , se for lega , traduzir o endereço e executar as instruções. O programa então é executado. Após isso , o timer interrupt é disparado , o modo é alterado para kernel e passamos para o interrupt handler. O handler decide para A e rodar B por meio da call switch(). Os registradores , estruturas e informações do PCB de A são salvos na process struct e os registradores , estruturas e informações do PCB de B são restauradas da process struct , a funcão return-from-trap e faz com que o processador retorne para executar B. O hardware restora os registradores de B , altero o modo para usuário e pula o PC de b , O processo b roda e executa um load out of bounds , o hardware checa o bound , altera para kernel mode e chama o trap-handler. O sistema operacional , de volta ao controle , decide por matar o processo B , desaloca sua memória e libera B da tabela de processos.

Capítulo 16 - Segmentation

1. Qual a razão para se utilizar segmentação?

Utilizando uma arquitetura da memória simples como a disposta em capítulos anteriores, pode-se notar uma grande quantidade de memória disperdiçada entre a Stack e Heap , que está , de alguma maneira ocupando também uma parte da memória física visto que caso a memória livre apenas não fosse alocada teriamos um problemas com os valores dos registradores bound. A partir desse problema a segmentação surge para resolver o problema referente a essa memória disperdiçada , segmentos lógicos são criados com o intuido de evitar o despercio de memória , e agora , dois registradores Base/Bounds por mmu são necessários.

2. Como se define qual o segmento utilizado a partir de um endereço dado?

Geralmente , para se definir qual segmento lógico está sendo utilizado , a partir de um endereço dado , utiliza-se a abordagem explicita (utilizada nos sistemas VAX/VMS), onde alocam-se alguns bits mais significativos do endereço para indicar o segmento referente. Para separar entre código , heap e stack precisariamos de dois bits , contudo , para evitar o desperdicío de um endereço binário , alguns sistemas colocam o código no mesmo segmento da Heap para poderem utilizar apenas um bit como indicação de segmento. Os bits restantes indicam o offset . Também deve-se citar a abordagem implícita , onde o hardware determina o segmento ao analisar como o endereço foi formado , se o endereço foi formado pelo PC , provavelmente se trata de um código , se o endereço se trata de um endereço gerado pela stack pointer ou base pointer , trata-se de um valor no segmenta da stack , qualquer outro valor é a heap.

Capítulo 17 – Free space managment

1. Quais as estruturas de dados utilizadas para gerenciar espaços livres em memória?

Geralmente , as estrutura de dados genéricas utilizadas para gerenciar espaços livres em memória são as chamadas listas de livres . Contém referências a todos os chunks de espaço livre na região administrada da memória. Não necessita de ser uma lista , mas apenas um estrutura de dados que armazene os espaços livres. Para mapear os espaços livres a lista deve conter basicamente o endereço de ínicia e tamanho desse espaço , caso uma memória seja solicitada o lista efetuará o chamado “spliting” , achar um chunk suficiente e dividi-lo em dois. Quando algum chunk é libertado , a lista de efetuar o chamado “colaescing” de espaço livre , quando um chunk é liberado , deve-se checar seus vizinhos , se são chunks livres também , deve-se fundi-los.

2. Em caso de utilização de lista de livres, onde esta pode ser armazenada? Como funciona este armazenamento?

Uma lista de livres deve ser armazenada no próprio espaço livre , como a função malloc precisa de uma lista para funcionar , ela não poderá ser utilizada. A lista é inicializada de forma que cada elemento terá um endereço , tamanho e apontará para o próximo elemento , para iniciar a lista deve utilizar a system call mmap() , utilizada para retornar um ponteiro para um espaço vazio. No caso de um espaço de 4KB teriamos um retorno de 4KB – Header. A partir dai teremos a lista inicializada com o espaço livre sendo referente a Heap inteira. A lista está criada , a partir desse ponto , quando ocorre uma solicitação via malloc , por exemplo , a lista será varrida (no ínicio com apenas um elemento) e as técnicas descritas anteriormente serão aplicadas para alocação dinâmica de memória. A lista só terá mais elementos , quando a função free for utilizada e tivermos algo semelhante a uma fragmentação , para reduzirmos o tamanho da lista devemos aplicar o conceito de coalesce.

3. Quais os algoritmos utilizados para se alocar um processo em um pedaço de memória? Como eles funcionam?

Os algoritmos utilizados para se alocar um processo devem buscar velocidade e a menor fragmentação possivel. Best Fit busca na lista de livres os chunks de memória do mesmo tamanho ou maiores que a memória solicitada e retorna o menor desses chunks. Worst fit , é basicamente o contrário da best fit , encontra o maior chunk disponível e retorna o espaço solicitado a partir dele. First fit , simplesmente encontra o primeiro bloco que satisfaz o tamanho do solicitante e o retorna. Next Fit , ao invés de sempre começar do começo da lista , mantem um ponteiro para o local que a lista estava quando fez sua última alocação e parte em busca do espaço a partir dai.Segregated list , caso uma aplicação tenha uma requisição que seja feita constantemente , uma lista é separada para atribuir objetos daquele tamanho específico , todas outras requisições são encaminhadas para uma lista mais geral. Buddy alocation , quando uma requisição de memória é feita , a busca por espaço livre recursivamente divide o espaço por dois até obter o menor bloco que possa alocar o espaço solicitado.