1. Os serviços e funções oferecidas por um sistema operacional podem ser divididas em duas categorias. Descreva brevemente as duas categorias e discuta como elas se diferem.

O sistema operacional pode agir como um **gerenciador de recurso** ou uma **máquina virtual**.

Se diz **gerenciador de recurso** o fato do SO administrar os recursos da máquina, tais como alocação de memória, I/O, o que vai ser executado na CPU, etc.

Como **máquina virtual** significa que o SO abstrai o hardware de um único computador em diversos ambientes de execução, dando a impressão ao aplicativo que ele está rodando em uma máquina somente para ele. Isso é possível usando técnicas de CPU Scheduling (escalonamento) e Memória Virtual.

2. Liste 5 (cinco) serviços, oferecidos por um sistema operacional, que são projetados para tornar o sistema de computação mais conveniente para os usuários.

1 - Operações de I/O;
2 - Manipulação de sistemas de arquivos;
3 - Comunicação entre processos;
4 - Execução de programas;
5 - Detecção de erros.

3. O que são System Calls, cite 4 exemplos.

Os processos devem utilizar as system calls quando precisam de um serviço do sistema. Com a System call estarão transferindo o controle da CPU para o sistema e este poderá realizar o serviço desejado.

Exemplos de System Calls: read, write, open, close, chmod.

4. Os sistemas operacionais podem ser construídos de diferentes maneiras. Descreva as principais arquiteturas existentes.

Monolitico: todos os serviços são executados em modo kernel, não tem um núcleo e sim um único bloco; é uma arquitetura de kernel onde todo o kernel é executado no espaço do kernel no modo de supervisão;

Micro-Kernel: a maioria dos serviços são executados em modo usuário, e possui um núcleo com o mínimo possível para o S.O.

Camadas: Estrutura modularizada que divide as operações em camadas. Cada uma delas realiza um conjunto de serviços e acessa apenas a camada imediatamente abaixo.

5. Descreva as ações tomadas pelo kernel para fazer a troca de contexto entre processos.

Primeiro o kernel armazena todos os registradores que o processo atual esta utilizando, em principal o contador de programa, e qualquer outro dado especifico do processo. Estas informações são armazenadas em um bloco de controle de processo o qual é situado na tabela de processos, então nesta mesma tabela é feita a restauração do bloco de dados do processo que será executado.

6. Explique o que são os anéis de execução. Qual a diferença entre código executando no nível 0 e em outros níveis.

São diferentes niveis de privilégios de execução de codigo para proteger dados e comportamento malicioso de programas. No nível zero qualquer instrução do processador

pode ser executada, e na medida em que o nível de proteção aumenta, algumas instruções e alguns acessos passam a não ser permitidos.
Exemplo:
Anel 3 – Usuário (Menor prioridade e maior proteção);
Anel 2 – Drivers (Prioridade maior e proteção menor que o 3);
Anel 1 – Drivers (Prioridade maior e proteção menor que o 2);
Anel 0 – Kernel (Maior prioridade de todos e sem proteção).

Sistemas operacionais como Windows e Linux normalmente utilizam apenas os anéis zero (Kernel Mode) para a execução de seus componentes internos e drivers, e o anel 3 (User mode) para os aplicativos de usuários desenvolvidos para esta plataforma. O número de aneis e suas caracteristicas podem variar de acordo com a arquitetura dos processadores.

7. O que são processos, e quais os estados que podem assumir?

O processo é basicamente um programa em execução. Este contem uma lista de posições de memoria e o espaço de endereçamento.

Estados do processo: novo, em execução, em espera, pronto (esperando para ser designado a um processador), terminado.

8. O que são threads? Em que diferem de processos convecionais?

Threads são linhas de execução de um processo. Um processo pode ter varias threads, quando possui mais de uma é chamado de multithreads.

As threads dentro de um processo compartilham das mesmas informações, só diferenciam-se no fato que cada uma tem sua pilha e seus registradores. As posições de memoria, dados e código são as mesmas. Portanto as trocas de contexto entre threads demandam bem menor uso da cpu.

8.1. As principais seções de um processo são Pilha, Heap, Dados e Código. Quais destas seções podem e/ou devem ser compartilhadas entre threads?

A heap, dados e código são compartilhados pelas Threads.

8.2. Qual a diferença de threads em nível de usuário e em nível de SO?

9. Mostre um exemplo de uso do fork(). Explique quais são os valores retornados pela função.

```
int main()
{
   int pid;

   pid = fork(); /* Ramifica um outro processo */
   if(pid == -1) /* erro */
   {
      perror("E impossivel criar um filho") ;
      exit(-1) ;
   }
   else if(pid == 0) /* filho */
```

```
    {
        execlp("/bin/ls", "ls", NULL);
    }
    else /* pai */
    {
        wait(NULL); /*o pai irá esperar pela conclusão do filho */
        exit(0);
    }
}
```

Valores de retorno: Zero para o processo filho criado corretamente, e o identificador do processo filho para o processo pai; -1 em caso de erro.

10. Na criação de processos utilizando fork() um novo processo é criado com a imagem do processo pai. Como o kernel Linux evita a necessidade de realizar esta cópia no momento da chamada do fork?

O kernel Linux possui uma system call chamada de execlp que realoca o espaço de memoria do processo para um novo programa.

COW - Copy on Write. Este método consiste em utilizar a mesma memória para o processo pai/filho, marcada como copy-on-write, e quando algum dado precisa ser alterado na memória, uma cópia desta página é feita separando os dados de cada processo.

11. O que significa escalonamento preemptivo?

O escalonamento é preemptivo quando o kernel tem a possibilidade de controlar o tempo que a CPU será utilizada por cada processo. De acordo com uma serie de prioridades um processo pode ser destituido do uso da cpu e outro processo receberá o controle. Este tipo de escalonamento permite a ilusão de paralelismo na execução dos processos, vendo que sem uma retirada forçada do processo de execução o mesmo poderia ficar em execução por horas antes de necessitar esperar algum periférico ou disco.

12. O que significa dizer que o Kernel também é preemptivo?

Quer dizer que as proprias chamadas de sistema do kernel estão sujeitas a preempção, isto poderia resultar em grandes problemas pois esta situação poderia gerar informações inconsistentes. Por isso o kernel possui algumas maneiras de evitar esses incidentes, como aguardar a finalização no caso de chamadas de sistemas ou blocos de I/O.

13. O que é starvation? Mostre um algoritmo que poderia levar a essa condição.

Um processo está em starvation ou bloqueio indefinido quando este se encontra pronto para executar, mas pela sua baixa prioridade este processo pode acabar sendo considerado bloqueado, ou seja, não será executado pois o sistema sempre terá processos de maior prioridade a serem antes executados.

Exemplo: uma impressora que tem como prioridade para impressão de arquivos o menor número de páginas. O arquivo que tiver o maior número de páginas corre o risco de nunca ser impresso caso o fluxo de impressão de arquivos com menos páginas for grande. Logo ele entra em starvation.

14. O que é um deadlock? Mostre um um algoritmo que pode entrar em deadlock.

Quando um processo solicita recursos ao sistema e estes já estiverem ocupados, o processo entra em estado de espera. Mas se o processo que está utilizando esse recurso

tambem estiver em estado de espera aguardando por outro recurso que tambem está preso a estados em espera. Então teremos um deadlock

Exemplo: Jantar dos filosofos; dois palitos e duas pessoas de frente uma pra outro, pega o palito da mão direita, e em seguida aguarda para pegar o palito da mão esquerda.

15. Quais dos seguintes algoritmos de escalonamento podem levar a "starvation" e porque?

a) First-come, Fist-served
b) Shortest job first
c) Round Robin
d) Priority

b) Shortest job first, pode sempre aparecer novos processos com jobs pequenos, o que iria impedir os de jobs grande de executar.

d) Priority, como ele trabalha alocando na CPU os processos de maior prioridade, um processo de baixa prioridade pode acabar bloqueado pois sempre terá outros processos com maior prioridade surgindo e utilizando a cpu antes desse processo que estará em starvation.

16. Que técnica é usada para evitar que um processo em um algoritmo de escalonamento por Multilevel Feedback-Queue nunca execute?

Para evitar o starvation o processo de escalonamento por Multilevel Feedback-Queue utiliza-se de filas de prioridade em que a alocação de processos é não estatica, onde um processo é inserido na fila e, mesmo que de baixa prioridade qualquer processo tenderá a se deslocar para uma fila de maior prioridade sempre que algum processo destas filas for finalizado.

17. O que deve ser considerando ao escrever um algoritmos de escalonamento em um sistema SMP?

18. Explique como funcionava o escalonador de prioridades no Kernel Linux (de 2.6 até 2.6.23)

19. Explique o algoritmo de escalonamento Completely Fair Scheduler que atualmente é utilizando no kernel Linux.

20. Mostre como funciona a solução de alternância estrita. Explique qual é sua limitação na prática.

Nesta solução o sistema obriga que cada processo utilize a região critica um por vez. Quer dizer, apos um processo utilizar a região critica este deverá aguardar que outro processo execute a região critica, então, o primeiro processo poderá voltar a executar nesta região. O problema na pratica é manter a alternancia precisa entre os processos e garantir que ambos tenham o mesmo numero de acessos.

21. O que é o problema da Região Crítica? Mostre um exemplo onde não tratá-lo poderia levar a um erro.

O problema da região critica é evitar que mais de um processo utilizarem dados compartilhados (as chamadas regiões criticas) ao mesmo tempo. Evitando assim que os processos obtenham dados inconsistentes.

22. Mostre e explique o funcionamento da solução de Peterson.

23. O que são semáforos? Mostre um exemplo de uso.

Semáforo é uma variável protegida que guarda um contador para controlar acesso a um determinado recurso compartilhado num ambiente multitarefa. O valor de um semáforo indica quantos processos ou threads podem ter acesso a um recurso compartilhado.

As principais operações sobre semáforos são:

- inicialização: valor inteiro indicando a quantidade de processos que podem acessar um determinado recurso.
- down: Decrementa o valor do semáforo. Se o semáforo está com valor zerado, o processo é posto para dormir.
- up: Se o semáforo estiver com o valor zero e existir algum processo adormecido, um processo será acordado. Caso contrário, o valor do semáforo é incrementado.

As operações de incrementar e decrementar devem ser operações atômicas, ou seja, enquanto um processo estiver executando uma dessas duas operações, nenhum outro processo pode executar outra operação sob o mesmo semáforo, devendo esperar que o primeiro processo encerre sua operação sob o semáforo. Essa obrigação evita condições de disputa entre vários processos.

Para evitar espera ocupada, que desperdiça processamento da máquina, a operação wait utiliza uma estrutura de dados. Quando um processo executa essa operação e o semáforo tem o seu valor zerado, este é posto na estrutura. Quando um outro processo executar a operação up e há processos na estrutura, uma delas é retirada.

Exemplo de uso: produtor-consumidor.

24. Mostre uma solução usando semáforos para o problema "Bounded-Buffer"

O problema do produtor-consumidor. O produtor coloca informação no buffer e o consumidor, retira a informação do buffer.

Problema: O produtor quer colocar informações no buffer e este esta cheio.
Solução: Colocar produtor pra dormir enquanto consumidor não tirar um ou mais itens do buffer.

Problema: O consumidor quer tirar informação e o buffer está vazio.
Solução: coloca o processo consumidor pra dormir.

```
#define N 100        /* número de posições */
int count=0;        /* número de itens no buffer */
```

```
void producer (void)
{
   int item;
   while (TRUE)              /* loop */
   {
      produce_item(&item);       /* geração do próximo item */
      if(count == N) sleep();       /* se buffer cheio, processo vai dormir /*
      enter_item(item);       /* colocação de um item no buffer */
      count = count +1;       /* incrementa a contagem de itens no buffer */
      if(count == 1) wakeup(consumer); /* o buffer está vazio */
   }
}
void consumer (void)
{
   int item;
   while(TRUE)                    /* loop */
   {
      if(count == 0) sleep();       /* se buffer vazio, processo vai dormir */
      remove_item(&item);       /* retirada de um item do buffer */
      count = count -1;       /* decrementa a contagem de itens do buffer */
      if (count == N-1) wakeup(producer); /* o buffer está cheio ? */
      consume_item(item);       /* imprime item */
   }
}
```

Condição de corrida: pode ocorrer por causa do acesso a count ser irrestrito. O buffer está vazio e o processo consumidor acabou de ler count para verificar se seu valor é 0. Neste momento ocorreu uma interrupção e o processo produtor começa a rodar, colocando um item no buffer e incrementando o count, cujo valor passa a ser 1. Baseado no fato do count estar em 0 antes do início de sua execução, o processo produtor infere que o processo consumidor deve estar dormindo, e então chama a primitiva WAKEUP para acordá-lo.

Infelizmente o processo consumidor não está dormindo, apenas interrompido. Logo o sinal emitido pelo WAKEUP, irá se perder pois não existe nenhum processo dormindo. Quando o escalonador selecionar o processo consumidor, este testará o count lido anteriormente que é zero e desta vez vai dormir. Mais cedo ou mais tarde o processo produtor vai encher o buffer e irá dormir eternamente.

Serão utilizados três semáforos:

- full para contar as posições do buffer que já foram preenchidas;
- empty para contar o número de posições vazias;
- mutex para assegurar que mais de um processo acesse o buffer ao mesmo tempo.

O valor inicial de full é 0, o de empty igual ao número de posições disponíveis no buffer, e mutex é inicialmente 1.

```
#define  N 100             /* número de posições do buffer */
typedef int semaforo; /* semaforo é um tipo especial representado por inteiros */
semaforo mutex =1; /* controla o acesso a região crítica */
semaforo empty = N; /* conta as posições vazias do buffer */
semaforo full = 0;     /* conta as posições ocupadas do buffer */
void producer (void)
```

```
{
   int item;
   while(TRUE)              /* TRUE é a constante 1 */
   {
      produce_item(&item);    /* gera algo para ser colocado no buffer */
      down(&empty);             /*decrementa o contador de posições vazias */
      down(&mutex);             /* entra na região crítica */
      enter_item(item);       /*coloca um novo item no buffer */
      up(&mutex);              /*deixa a região crítica */
      up(&full);             /*incrementa o contador de posições ocupadas */
   }
}

void consumer (void)
{
   int item;
   while(TRUE)                  /* loop */
   {
      down(&full);             /*decrementa o contador de posições ocupadas */
      down(&mutex);          /*entra na região crítica */
      remove_item(&item);     /*retira item do buffer */
      up(&mutex);              /*deixa a região crítica */
      up(&tah);             /*incrementa o contador de posições vazias */
   }
}
```

A utilização do semáforo serviu para evitar a condição de corrida através da exclusão mútua.

25. Mostre uma solução usando semáforos para o problema "Readers-Writers"

Existem várias soluções.
http://en.wikipedia.org/wiki/Readers-writers_problem

26. Mostre uma solução usando semáforos para o problema "Dining-Philosophers"

Existem várias soluções.
http://www.isi.edu/~faber/cs402/notes/lecture8.pdf
http://www.math-cs.gordon.edu/courses/cs322/projects/p2/dp/
http://en.wikipedia.org/wiki/Dining_philosophers_problem

27. Resolva pelo menos 3 problemas de sincronização do Livro "The Little Book of Semaphores".

Outras:

O que é copy-on-write e sob quais circunstâncias ela é útil?

É uma estratégia de otimização que evita que um determinado dado seja copiado por completo desnecessariamente. É utilizado principalmente em sistemas de memória virtual, onde um processo é dividido em dois, mas a memória de cada um dos processos permanece a mesma enquanto o conteúdo delas não for alterado após a separação.

Explique o que é segmentação e paginação.

(precisa mais informações) São técnicas utilizadas para proteger a memória. Um segmento é representado por um offset e um length, contendo também as suas permissões. Caso seja feito um acesso fora do segmento ou sem permissões de acesso, uma exceção é lançada. Com paginação, toda a memória é vista como várias páginas de tamanho definido e fixo. Uma página tem definido seus controles de acesso, escrita e/ou execução.

http://www.las.ic.unicamp.br/edmar/P**otimização que evita que um determinado dado seja copiado por completo desnecessariamente. É ut**UC/2006/SO/SO-Aula6.pdf

Explique o que é endereço lógico, endereço linear e endereço físico. Qual unidade (paginação ou segmentação) é responsável por converter cada um destes endereços.

Logical address
Included in the machine language instructions to specify the address of an operand or of an instruction. This type of address embodies the well-known 80 x 86 segmented architecture that forces MS-DOS and Windows programmers to divide their programs into segments . Each logical address consists of a segment and an offset (or displacement) that denotes the distance from the start of the segment to the actual address.

Linear address (also known as virtual address)
A single 32-bit unsigned integer that can be used to address up to 4 GB that is, up to 4,294,967,296 memory cells. Linear addresses are usually represented in hexadecimal notation; their values range from 0x00000000 to 0xffffffff.

Physical address
Used to address memory cells in memory chips. They correspond to the electrical signals sent along the address pins of the microprocessor to the memory bus. Physical addresses are represented as 32-bit or 36-bit unsigned integers.
Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-2-sect-1.html#idx-chp-2-0297

A segmentação no Kernel Linux não é usada intensamente, porque? Também diga quais são os 4 principais segmento utilizados neste sistema operacional.

Como consequência direta da segmentação, não é necessário saber de antemão onde a aplicação está sendo executada na memória, facilitando tremendamente o trabalho dos programadores, por exemplo.
Isto é definido em estruturas internas do kernel. Em linux, no entanto, a segmentação é um pouco mais complicada. Linux utiliza apenas 4 segmentos para todos os seus processos. 2
segmentos para o KERNEL SPACE de [0xC000 0000] (3GB) até [0xFFFFFFFF] (4GB) e 2 para o USER SPACE de [0] até [0xBFFF FFFF](3GB). São eles:

1. Kernel Code
2. Kernel Data / Stack
3. User Code
4. User Data / Stack

No USER SPACE, um processo não pode exceder 3GB. Isto significa que apenas as primeiras 768 páginas -conceito abordado à seguir -são significativas 768x4096KB =3GB.
O problema com a segmentação é que o processo de alocação de memória não é dinâmico no sentido de que os segmentos são áreas contínuas de memória e não podem ser partidas. Isto significa que para se carregar um novo processo temos que garantir que existe um

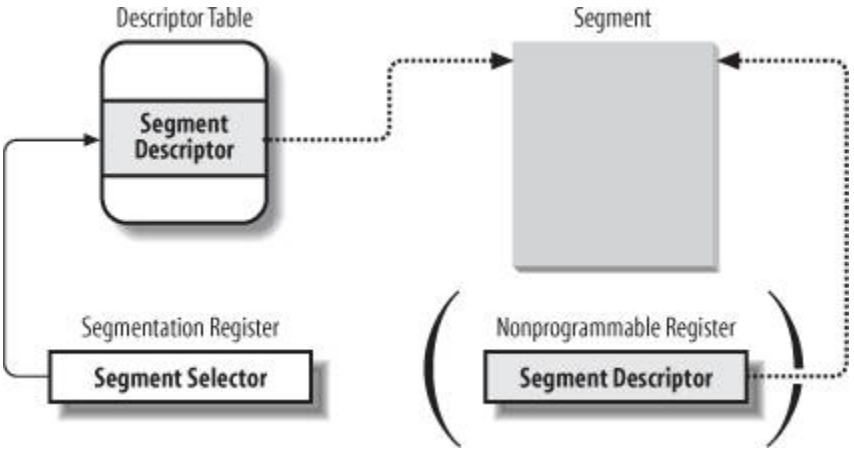
segmento disponível e que o tamanho deste segmento comporte o dado processo. Caso contrário, segmentation fault, ou seja, não existe um segmento disponível para alocar aquele processo.

Referencia:
http://www.slackware-brasil.com.br/web_site/artigos/artigo_completo.php?aid=3533

No kernel Linux executando na arquitetura x86 como é feito o Acesso Rápido aos Descritores de Segmento?

We recall that logical addresses consist of a 16-bit Segment Selector and a 32-bit Offset, and that segmentation registers store only the Segment Selector.
To speed up the translation of logical addresses into linear addresses, the 80 x 86 processor provides an additional nonprogrammable registerthat is, a register that cannot be set by a programmerfor each of the six programmable segmentation registers. Each nonprogrammable register contains the 8-byte Segment Descriptor (described in the previous section) specified by the Segment Selector contained in the corresponding segmentation register. Every time a Segment Selector is loaded in a segmentation register, the corresponding Segment Descriptor is loaded from memory into the matching nonprogrammable CPU register. From then on, translations of logical addresses referring to that segment can be performed without accessing the GDT or LDT stored in main memory; the processor can refer only directly to the CPU register containing the Segment Descriptor. Accesses to the GDT or LDT are necessary only when the contents of the segmentation registers change (see Figure 2-4).

Figure 2-4. Segment Selector and Segment Descriptor



Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-2-sect-2.html#idx-chp-2-0321 [Veja o item 2.2.3**]**

Explique a diferença entre fragmentação interna e externa.

Fragmentação externa: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-8-sect-1.html#idx-chp-8-2152
frequent requests and releases of groups of contiguous page frames of different sizes may lead to a situation in which several small blocks of free page frames are "scattered" inside blocks of allocated page frames.
Fragmentação interna: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-8-sect-2.html#idx-chp-8-2178

A better approach instead consists of introducing new data structures that describe how small memory areas are allocated within the same page frame. In doing so, we introduce a new problem called internal fragmentation. It is caused by a mismatch between the size of the memory request and the size of the memory area allocated to satisfy the request.

Que esquema de paginação é utilizado para mapear mais que 4GB na arquitetura x86 de 32 bits?

Physical Address Extension (**PAE**)

Chapter: 2.4. Paging in Hardware

Ref1: http://book.opensourceproject.org.cn/kernel/kernel3rd/index.html?page=opensource/0596005652/understandlk-chp-2-sect-1.html

Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-2-sect-1.html#idx-chp-2-0297

Dadas partições de memória de 100KB, 500KB, 200KB, 300KB e 600KB (em ordem), como cada um dos algoritmo: First-fit, best-fit e worst-fit, alocariam processos de 212Kb, 417Kb, 112KB e 426KB? Qual dos algoritmos faria o uso mais eficiente de memória.

Explique os seguintes algoritmos para substituição de páginas (page replacement): FIFO, Optimal e LRU. No caso do LRU, você deve explicar quais são as políticas de aproximação para o LRU.

FIFO – associa a cada página a hora que a página foi conduzida à memória, quando necessário substituir é escolhido a página mais antiga presente na memória. Pode ser feito com uma fila.

Optimal – Substitui a página que não será utilizada pelo período de tempo mais longo. Difícil de implementar pois requer o conhecimento futuro da sequência de referêcnias, assim se uma sabendo da possível utilização de uma página no futuro, essa página é mantida em memória e escolhe para ser retirada de memória uma página com um futuro não promissor de uso.

LRU – Considerando o passado recente como uma aproximação para o futuro próximo, substituiremos a página que não foi usada pelo período de tempo mais longo, esta abordagem é do algoritmo do menos recentemente utilizado. Pode ser implementado com contadores ou uma pilha.

-substituição por aproximação ao LRU: com um bit de referência indica a utilização das páginas.

-bits de referência adicionais: um byte de refência que vai incrementando mostrando os acessos contínuos a uma página, mostrando seu histórico, quanto maior esse byte mais recentemente foi utilizado.

-segunda chance: verifica o bit de referência, se o valor for 0 substitui essa página, se for 1 dá àquela página uma segunda chance, seu bit é desligado e sua hora de chegada é reposicionada para a hora corrente.

-segunda chance estendido: usa dois bits para chance: (0,0) nem recentemente utilizada nem modificada, melhor para substutuir; (0,1) não recentemente usada mas modificada, tem que ser guardada em disco antes de substituída; (1,0) recentemente utilizada mas não modificada, provavelmente será utilizada em breve; (1,1) recentemente utilizada e modificada, provavelmente será utilizada novamente em breve e terá que ser gravada em disco antes de ser substituída.

Como funciona o mapeamento de arquivos em memória, qual a vantagem de utilizar esse modo de acesso.

Finally, Linux supports the mmap() system call, which allows part of a file or the information stored on a block device to be mapped into a part of a process address space. Memory mapping can provide an alternative to normal reads and writes for transferring data. If the same file is shared by several processes, its memory mapping is included in the address space of each of the processes that share it.

O que é o modo de alocação de memória "Buddy System". Qual a sua desvantagem?

The technique adopted by Linux to solve the external fragmentation problem is based on the well-known buddy system algorithm. All free page frames are grouped into 11 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 contiguous page frames, respectively. The largest request of 1024 page frames corresponds to a chunk of 4 MB of contiguous RAM. The physical address of the first page frame of a block is a multiple of the group sizefor example, the initial address of a 16-page-frame block is a multiple of 16 x 2^{12} (2^{12} = 4,096, which is the regular page size).

We'll show how the algorithm works through a simple example:

Assume there is a request for a group of 256 contiguous page frames (i.e., one megabyte). The algorithm checks first to see whether a free block in the 256-page-frame list exists. If there is no such block, the algorithm looks for the next larger blocka free block in the 512-page-frame list. If such a block exists, the kernel allocates 256 of the 512 page frames to satisfy the request and inserts the remaining 256 page frames into the list of free 256-page-frame blocks. If there is no free 512-page block, the kernel then looks for the next larger block (i.e., a free 1024-page-frame block). If such a block exists, it allocates 256 of the 1024 page frames to satisfy the request, inserts the first 512 of the remaining 768 page frames into the list of free 512-page-frame blocks, and inserts the last 256 page frames into the list of free 256-page-frame blocks. If the list of 1024-page-frame blocks is empty, the algorithm gives up and signals an error condition.

The reverse operation, releasing blocks of page frames, gives rise to the name of this algorithm. The kernel attempts to merge pairs of free buddy blocks of size b together into a single block of size 2b. Two blocks are considered buddies if:

- Both blocks have the same size, say b.
- They are located in contiguous physical addresses.
- The physical address of the first page frame of the first block is a multiple of 2 x b x 2^{12}.

The algorithm is iterative; if it succeeds in merging released blocks, it doubles b and tries again so as to create even bigger blocks.

Desvantagem seria criar fragmentação interna?

"...However, because of the way the buddy memory allocation technique works, there may be a moderate amount of internal fragmentation - memory wasted because the memory requested is a little larger than a small block, but a lot smaller than a large block."

Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-8-sect-1.html#idx-chp-8-2095 [Veja o item 8.1.7]

O que é o mecanismo de alocação chamado SLAB?

Having a slab cache has some other advantages besides reducing waste.

Allocation and freeing is faster than with the buddy system.

There is an attempt at doing some *cache colouring*.

Referência: http://book.opensourceproject.org.cn/kernel/kernhttp://74.125.93.132/search?q=cache:gVSf51lIh3kJ:www.ravel.ufrj.br/arquivosPublicacoes/MemoryAddressing.pdf+segmentação+em+linux&cd=2&hl=pt-BR&ct=clnk&gl=br&client=firefox-ael3rd/opensource/0596005652/understandlk-chp-8-sect-2.html#idx-chp-8-2197

O que é o Overcommit de memória? Quais as possíveis configurações

para a váriavel de kernel /proc/sys/vm/overcommit_memory no kernel 2.6?
http://linux-mm.org/OverCommitAccounting
Controls overcommit of system memory, possibly allowing processes to allocate (but not use) more memory than is actually available.

- 0 - Heuristic overcommit handling. Obvious overcommits of address space are refused. Used for a typical system. It ensures a seriously wild allocation fails while allowing overcommit to reduce swap usage. root is allowed to allocate slighly more memory in this mode. This is the default.
- 1 - Always overcommit. Appropriate for some scientific applications.
- 2 - Don't overcommit. The total address space commit for the system is not permitted to exceed swap plus a configurable percentage (default is 50) of physical RAM. Depending on the percentage you use, in most situations this means a process will not be killed while attempting to use already-allocated memory but will receive errors on memory allocation as appropriate.

Referência do Documentation/sysctl/vm.txt:
http://www.mjmwired.net/kernel/Documentation/sysctl/vm.txt

```
overcommit_memory:

This value contains a flag that enables memory overcommitment.

When this flag is 0, the kernel attempts to estimate the amount
of free memory left when userspace requests more memory.

When this flag is 1, the kernel pretends there is always enough
memory until it actually runs out.

When this flag is 2, the kernel uses a "never overcommit"
policy that attempts to prevent any overcommit of memory.

This feature can be very useful because there are a lot of
programs that malloc() huge amounts of memory "just-in-case"
and don't use much of it.

The default value is 0.

See Documentation/vm/overcommit-accounting and
security/commoncap.c::cap_vm_enough_memory() for more information.
```

No kernel Linux qual é a chamada de sistema utilizada para aumentar o heap do processo? Qual sua relação com a função malloc()?
brk() (não é uma chamado ao sistema, é somente um bilbioteca em C empacotada)
Os dois server para aumentar a quantidade de memória?
brk() é syscall sim! E é ela que é utilizada para aumentar o heap do processo.
malloc() é implementado com brk()'s (o default do linux usa análises para definir se o malloc chama uma simples brk ou mmap em determinado momento).
Em que momentos é executado o algoritmo de "Page Frame Reclaiming"?
Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-17-sect-1.html#idx-chp-17-4083
PFRA = Page Frame Reclaiming Algorithm

Thus, sooner or later all the free memory will be assigned to processes and caches. The page frame reclaiming algorithm of the Linux kernel refills the lists of free blocks of the buddy system by "stealing" page frames from both User Mode processes and kernel caches.

Actually, page frame reclaiming must be performed before all the free memory has been used up. Otherwise, the kernel might be easily trapped in a deadly chain of memory requests that leads to a system crash. Essentially, to free a page frame the kernel must write its data to disk; however, to accomplish this operation, the kernel requires another page frame (for instance, to allocate the buffer heads for the I/O data transfer). If no free page frame exists, no page frame can be freed.

One of the goals of page frame reclaiming is thus to conserve a minimal pool of free page frames so that the kernel may safely recover from "low on memory" conditions.

Considerando algoritmo de "Page Frame Reclaiming" o que acontece quando os processos de usuários começam a consumir mais memória?

Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-17-sect-1.html#idx-chp-17-4083

Qual a relação entre o espaço de endereçamento de um processo e a utilização da memória física?

Referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-1-sect-6.html#idx-chp-1-0280

Veja item 1.6.4

Note que o espaço de endereçamento do processo é linear o que não é refletido na memória física.

Outra referência: http://book.opensourceproject.org.cn/kernel/kernel3rd/opensource/0596005652/understandlk-chp-9-sect-1.html

No Linux o que significa dizer o seque a memória virtual de um processo é 50MB, a memória residente 20MB e a memória compartilhada 10MB.

OBSERVAÇÕES

Para toda a parte de memória (slab, overcommit, buddy system) vejam essa referência, é uma explicação mais simples e concisa que pode ser útil:
http://www.win.tue.nl/~aeb/linux/lk/lk-9.html

Informações sobre HighMemory: http://linux-mm.org/HighMemory

Um bom "resumo": http://www.ibm.com/developerworks/linux/library/l-linux-kernel/

Estrutura da tabela de páginas (por arquitetura): http://linux-mm.org/PageTableStructure

Documento em português sobre Segmentação/Paginação: http://alumni.ipt.pt/~paulofer/cap7.html