**UNIVERSIDADE ESTADUAL DE PONTA GROSSA**

**SETOR DE CIÊNCIAS AGRÁRIAS E DE TECNOLOGIA**

**DEPARTAMENTO DE INFORMÁTICA**

**ENGENHARIA DE COMPUTAÇÃO**

**MATEUS FELIPE DA SILVA JUNGES**

**OSÉIAS MAGALHÃES**

# ATIVIDADES SISTEMAS OPERACIONAIS

**PONTA GROSA**

**2018**

1. **Caracterize os sistemas operacionais quanto à sua evolução histórica.**

   **R:** No início (1940), o programador era quem fazia o controle da execução da máquina, e acessava diretamente seus periféricos. No máximo, existia uma biblioteca com rotinas de I/O já programadas. A primeira modificação foi a utilização de operadores profissionais. O programado entrega um job, formado pelo programa a ser compilado e executado, preparados em cartões perfurados.

   Para diminuir o tempo de preparação para execução de diferentes jobs, agrupavam-se um lote de jobs com necessidades semelhantes. Isso ficou conhecido como sistema em batch.

   Em 1950 surgiram os monitores residentes, que é um programa que fica o tempo todo na memória. Quando um programa termina, ele avisa o monitor. Assim, o tempo que o computador fica parado diminui. Para que o monitor consiga executar suas tarefas, ele precisa saber o que fazer. Os cartões de controle fornecem essa informação. Esses cartões são, basicamente, identificadores para o início de execução de um job, ou para informar o monitor quando o programa finalizou sua execução.

   Em 1960, surge o conceito de multiprogramação. A ideia foi manter diversos programas na memória principal. Quando um programa está esperando a conclusão da entrada ou saída, outro programa inicia sua execução. Aqui, o tempo do processador é dividido entre diversos programas, o que faz com que ele seja melhor ocupado e fique menos tempo parado, ocupando melhor o hardware.

   Duas inovações em hardware possibilitaram o desenvolvimento de multiprogramação. O uso de interrupções e os discos magnéticos.

   A partir disso o computador passou a ler os jobs a partir de fita magnética, e não de cartões perfurados, tornando esse procedimento mais rápido. Como era necessário ler cartões de diferentes jobs, a execução não podeiria ser seqüencial. Daí a necessidade do uso dos discos magnéticos, facilitando a alternância entre jobs.

   Na década de 1960, iniciaram-se as experiências com timesharing, e em 1970 ocorreu sua disseminação. Em um ambiente

com multiprogramação, diversos programas dividem o tempo do processador. Em um sistema timesharing, além da multiprogramação, cada usuário possui um terminal.

A medida em que o hardware foi melhorando, o mesmo aconteceu com os sistemas operacionais. Atualmente, temos em microcomputadores mecanismos encontrados, algum tempo atrás, apenas em máquinas de grande porte.

Historicamente, SOs preocupam-se com principalmente com a eficiência no uso do computador. Atualmente, existe uma grande preocupação com a conveniência de uso.

Os sistemas operacionais estão em constante evolução. Uma das áreas de pesquisa mais importantes são os sistemas operacionais distribuídos, onde vários computadores estão interconectados por meio de uma rede de comunicação de algum tipo. Outra área importante são os sistemas operacionais de tempo real, onde os resultados devem estar corretos não somente do ponto de vista lógico, mas devem também ser gerados no momento correto.

2. **Cite 5 características de sistemas operacionais de tempo compartilhado.**
    - Permitem que diversos programas sejam executados a partir da divisão em pequenos intervalos, denominados de fatia de tempo.
    - Permitem a interação do usuário com o sistema, através de terminais que incluem vídeo, teclado e mouse.
    - O sistema cria para cada usuário um ambiente de trabalho próprio, dando a impressão de que todo o sistema está dedicado exclusivamente a ele.
    - Oferecem tempos de resposta razoáveis e custos mais baixos, em função da utilização compartilhada dos diversos recursos do sistema.

- Possuem uma linguagem de controle que permite ao usuário comunicar-se diretamente com o sistema operacional, através de comandos.

3. **O que é processo. Cite 6 elementos básicos que os representam.**

   **R:** Processo é um módulo executável único, que ocorre concorrentemente com outros módulos executáveis. Existem processos CPU bound, processos que utilizam muito o processador, em que o tempo de execução é definido pelos ciclos de clock, I/O bound, orientados a entrada e saída, que realizam muitas operações de entrada e saída de dados, em que o tempo de execução é definido pela duração destas.

   Características:

   - Código (instruções)
   - Conjunto de dados (variáveis)
   - Stack
   - Identificação única, com seu PID
   - Limite de recursos disponíveis no sistema
   - Acesso restrito ao sistema operacional e a outros processos

4. **O que é thread (processo leve). Cite 6 elementos básicos que as caracterizam.**

   **R:** É um pequeno programa que trabalha como um subsistema, sendo uma forma de um processo se autodividir em duas ou mais tarefas. Essas tarefas podem ser executadas em um único bloco ou praticamente juntas, mas que são tão rápidas que parecem estar trabalhando em conjunto e ao mesmo tempo.

   - Existe dentro de um processo
   - Usa recursos do processo
   - Tem seu próprio fluxo de controle independente enquanto existir o processo pai e o SO dá suporte a ele
   - Pode compartilhar os recursos do processo com outros threads igualmente independentes
   - Morre se o processo que a contém morre.

- Um thread mantém seu próprio PC, stak pointer, registradores, propriedades de escalonamento, conjunto de sinais pendentes e bloqueados e os dados específicos a uma thread.

5. **Os sistemas operacionais são construídos de acordo com uma organização interna (estrutura). Conceitue os modelos:**

   a) **Monolítico:**

   Nesse tipo de estrutura, o SO é escrito como uma coleção de rotinas, onde cada rotina pode chamar qualquer outra rotina, sempre que for necessário. Portanto, o sistema é estruturado de forma que as rotinas podem interagir livremente umas com as outras. Quando essa técnica é usada, cada rotina no sistema possui uma interface bem definida em termos de parâmetros e resultados.

   Os serviços do SO são implementados por meio de System Calls, em diversos módulos, executando em modo núcleo.

   Como todos os módulos executam no mesmo espaço de endereçamento, um bug em um dos módulos pode derrubar o sistema inteiro. Toda rotina é visível a qualquer outra.

   b) **Distribuído:**

   Crescimento incremental: um sistema distribuído apresenta facilidades de expansão, ao contrário de outro sistema que não permitem expansão, a não ser por repetição. A repetição apresenta o inconveniente de produzir dois ou mais sistemas distintos, em vez de um sistema maior;

   Confiabilidade: sistemas distribuídos podem ser potencialmente mais confiáveis devido à multiplicidade e a um certo grau de autonomia de suas partes. É notório que a distribuição física não é tão importante quanto a distribuição lógica. Esta última pode ser implementada tanto a um único processador quanto a vários processadores localizados num mesmo ambiente ou em ambiente distintos. A distribuição física está mais ligada a questões de

desempenho, tempo de resposta, organização do sistema e principalmente à possibilidade de ter-se controle explicito sobre o processamento e informações locais.

Estrutura: sistemas distribuídos podem refletir a estrutura organizacional à qual eles servem;

Proteção: sistemas distribuídos podem oferecer mais segurança do que sistemas centralizados. A segurança resulta muito mais da distribuição lógica do sistema do que da sua distribuição física.

**c) Camadas:**

divisão em camadas representa um agrupamento ordenado de funcionalidades, sendo que: (a) a funcionalidade específica de aplicativo está localizada nas camadas superiores, (b) a funcionalidade que abrange os domínios de aplicativos se encontra nas camadas intermediárias e (c) a funcionalidade específica do ambiente de implantação está nas camadas inferiores.

O número e a composição das camadas dependem da complexidade do domínio do problema e do espaço para solução:

Em geral, há apenas uma única camada específica de aplicativo.

Nos domínios em que sistemas anteriores foram desenvolvidos ou sistemas de grande porte são compostos de sistemas interoperacionais menores, há uma grande necessidade de compartilhar as informações entre as equipes de design. Conseqüentemente, para maior clareza, é provável que a camada específica de negócios exista parcialmente e esteja estruturada em várias camadas.

Os espaços para solução que são suportados por produtos de middleware e nos quais o software de sistema complexo desempenha um papel importante terão camadas inferiores bem

desenvolvidas, talvez com várias camadas de middleware e software de sistema.

Os subsistemas devem ser organizados em camadas com subsistemas específicos de aplicativo localizados nas camadas superiores da arquitetura, subsistemas específicos de operação e hardware nas camadas inferiores da arquitetura, e serviços para fins genéricos nas camadas de middleware.

**d) Máquina virtual:**

**e) Microkernel:**

É um núcleo minusculo que trabalha somente com o mínimo de processos possíveis, essenciais para manter o sistema em funcionamento, executando-os no kernel space. Todos os demais processos são executados por daemons conhecidas como servidores de forma isolada e protegidos no user space.

O sistema fica dividido mais ou menos assim: Servidor I/O , servidor de memória, servidor de gerenciamento de processos, servidor de sistema de arquivos, servidor de device drivers e etc. Esses servidores se comunicam com o microkernel; o sistema monitora continuamente cada um destes processos e se uma falha for detectada, ele substitui automaticamente este processo defeituoso sem reiniciar a máquina (reboot), sem perturbar os outros processos em execução e, principalmente, sem que o usuário perceba.

**6. Conceitue os modelos:**

**a) SISD**

SISD é a representação no esquema de Flynn do clássico computador de von Neumann. Ele possui apenas um fluxo de instrução e de dados, de modo que apenas uma operação é feita por vez, ou seja, é sequencial.

**b) SIMD**

Máquinas desse tipo possuem apenas um fluxo de instrução, mas possuem múltiplas unidades de cálculo. Isso significa que a máquina é capaz de executar uma mesma instrução em um conjunto de dados de maneira simultânea.

Existe um exemplo clássico para esse tipo: os processadores vetoriais, mas estes estão cada vez mais raros. Todavia o conceito está mais vivo do que nunca, presente nos processadores convencionais através do uso das instruções SSE dos processadores modernos (do pentium III pra frente), além das placas de vídeo que são especializadas neste tipo de operação --- uma operação feita em uma imagem é uma mesma operação feita em vários dados (pontos diferentes da imagem).

**c) MISD**

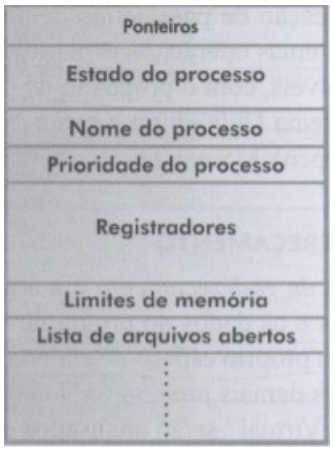
Máquinas MISD operam várias instruções diferentes em um único dado. É quase que um modelo completamente teórico, sem nenhum exemplo real (embora algumas pessoas usem o pipeline de instruções como exemplo de máquina MISD).

**d) MIMD**

São as máquinas que possuem CPUs independentes, onde cada unidade processante atua com instruções diferentes em dados diferentes. Os processadores com mais de um núcleo caem nesse modelo.

**7. Conceitue PCB. Enumere seus elementos.**

O bloco de controle de processo é uma estrutura de dados que serve para armazenar a informação necessária para tratar um determinado processo. Em alguns SOs o PCB é alocado no início da pilha do núcleo do processo, já que é uma localização protegida.




8. **Qual a finalidade da hierarquia de processos? Como os sistemas operacionais fazem o uso dela?**

   Subprocessos são processos criados dentro de uma estrutura hierárquica. Neste modo, o processo criador é chamado de processo pai e o subprocesso criado é chamado de processo filho. Os subprocessos podem criar novos subprocessos e assim por diante. Outra característica é a dependência existente entre o processo pai e o processo filho, quando o processo pai deixa de existir, todos os processos filhos também são eliminados automaticamente. Além desta característica, o processo pai pode compartilhar suas quotas com os processos filhos, assim, quando um subprocesso é criado, o processo pai cede parte de suas quotas para o processo filho.

9. **Relacione as funções que retornam ID de processos.**
   **getpid():** Retorna o Id do processo, no formato pid_t
   **getppid():** Retorna o ID do processo pai

10. **Explique o funcionamento de três funções da família EXEC.**
    **execl():** Recebe argumentos em argv, que serão listados um a um como parâmetros de função em forma de string.
    **execv():** Os argumentos que serão recebidos em argv são passados em um vetor do tipo char* que já contém todas as strings previamente carregadas.

**execlp():** Esta função irá buscar a nova imagem do processo nos diretórios contidos na variável PATH. Para as versões sem a letra "*p*", deverá ser passada uma *string* contendo o caminho completo para o arquivo executável.

**11. Explique o funcionamento do procedimento de inicialização de um computador.**

Quando o botão de ligar é pressionado, a fonte leva eletricidade para a placa-mãe, que em seguida ativa o processador e o cooler.

Logo após, quem entra em ação é o BIOS (Basic Input/Output System — Sistema Básico de Entrada/Saída em português), um sistema operacional pré-gravado no chipset que garante a tradução dos códigos de hardware para a tela — sua interface de configuração (Setup Utillity) é azul, sendo facilmente reconhecida (o que não quer dizer que seja facilmente entendida!) por muitos usuários.

Imagine que o BIOS sempre será o primeiro a acordar e a trabalhar assim que você põe o PC para funcionar. É ele que passa as primeiras ordens para o processador, além de verificar quais itens estão instalados na máquina.

O BIOS também é responsável por carregar a memória RAM, placa de vídeo, teclado, cachê básico e, por fim, possibilitar a inicialização do sistema operacional. Acompanhe em ordem cronológica as etapas que ele percorre:

1. Acessa a memória CMOS, um circuito integrado que grava informações referentes ao hardware. Nela, o BIOS estabelece reconhecimento e comunicação com peças como placas de vídeo e memória RAM.

2. A segunda fase, conhecida como Power-on Self Test (POST) nada mais é do que um conjunto de teste que a BIOS realiza para

saber se tudo está se inicializando da maneira correta. Quando alguns componentes essenciais estão faltando, alguns beeps ou mensagens na tela alertam o usuário.

3. A etapa seguinte consiste na procura de alguma fonte para inicializar o sistema operacional. Tal fonte é configurável e pode ser um disco rígido (padrão), CD-ROM, pendrive, disquete, entre outros.

4. Agora, o BIOS lê o setor zero (que contém apenas 512 bytes, denominado Master Boot Record) do HD. Essa área contém um código que alavanca a inicialização do sistema operacional. Outros dispositivos de boot (CDs, disquetes etc.) têm a capacidade de emular esse setor zero.

5. No caso do Windows, o Master Boot Record (MBR) verifica qual partição do HD está ativa (configurada como Master) e inicializa o "setor um" dela — essa área tem um código com a simples missão de carregar o setor dois.

6. A etapa seguinte consiste na leitura de um arquivo de configuração de boot, o Boot Loader (quando falamos do Windows, trata-se do NTLDR).

7. A partir dele, é inicializado o núcleo (kernel). Assim como o BIOS estabelece a ligação entre hardware e sistema, o kernel serve para firmar uma comunicação estável entre hardware e software. Nessa fase, é ele quem assume o controle do computador.

8. O kernel carrega os arquivos principais e informações básicas do sistema operacional (incluindo o registro), além de relacionar os componentes de hardware com as respectivas DLLs e drivers.

9. No entanto, o kernel não carrega todos os processos para não sobrecarregar o sistema — somente as operações essenciais são colocadas em atividade para possibilitar o início do Windows.

10. A tela de escolha de usuários é exibida e, após o logon, os programas relacionados para começar junto com o sistema são carregados.

**12. Como funciona o mecanismo de interrupções na arquitetura Intel?**

- O controlador de interrupções ativa o sinal INT
- A CPU responde com INTA (INTerrupt Acknowledge);
- O controlador de interrupções responde através do bus de dados ($D_0$ a $D_7$) com o número da entrada que produziu a interrupção;
- A CPU utiliza esse número para indexar uma tabela de endereços de memória (designados interrupt vectors) onde estão os conjuntos de instruções que servem cada interrupção.

**13. O que significa reentrância de código? Quais suas implicações?**

R: Código reentrante é um código que pode ser utilizado por diversos programas /processo, e, portanto, pode existir apenas uma cópia na memória. Pode ser interrompido no meio de execução e depois chamada novamente sem nenhum problema.

**14. O que significa e como evitar a região crítica.**

R: Região crítica é a parte do código que executa alteração no recurso compartilhado entre processos. Pode ser evitada utilizando-se de exclusão mútua, que tem por regras:

- Apenas um processo executa sua seção crítica por vez;

- Nenhum processo pode esperar indefinidamente para entrar na execução da seção crítica;

- Nenhum processo que não está na seção crítica pode bloquear a execução de outros processos;

- Não pode levar em conta o numero de processadores e suas velocidades relativas;

**15. Explique o funcionamento dos mecanismos de exclusão mútua:**

a) **Semáforos:** Em 1965 Dijkstra apresentou a ideia para resolver o problema produtor-consumidor com o uso de variáveis inteiras para monitorar o número de sinais *wakeup* para utilização futura. Tais variáveis ficaram conhecidas como **semáforos**, e sobre as mesmas foram estabelecida duas operações diferentes: P (Down) e V (Up), que é uma forma generalizada das operações *sleep*e *wakeup*.

Exemplo de funcionamento da operação de um semáforo:

**Operação** P(X) ou Down(X)
Verifica se o valor do semáforo X é positivo (X>0).
Caso afirmativo decrementa X de uma unidade (ou seja, consome um sinal de *wakeup*).
Caso negativo envia um sinal de **sleep**, fazendo inativo o processo.

**Operação** V(X) ou Up(X)
Incrementa o valor do semáforo X de uma unidade.
Existindo processos inativos, na ocorrência uma operação **down**, um deles é escolhido aleatoriamente pelo sistema para ser ativado (semáforo retornará a zero, i.e., X=0).

b) **Sleep/ Wakeup:** No problema típico produtor-consumidor dois processos compartilham um *buffer*, que é uma área de dados de tamanho fixo que assemelha-se a um reservatório temporário. O processo produtor coloca informações no *buffer* enquanto o processo consumidor as retira de lá.

Se forem processos sequenciais a solução do problema é simples, porém se forem processos paralelos aparecem então uma disputa de concorrência.
Quando o produtor não pode colocar informações no *buffer* porque ele já está cheio, ou o consumidor não pode retirar informações do *buffer*

porque ele está vazio. Nestes casos tanto o produtor como o consumidor poderiam ser *adormecidos*, isto é, ter sua execução suspensa, até que existisse espaço no *buffer* para que o produtor coloque novos dados ou existam dados no *buffer* para o consumidor poder retira-los. Um teste de solução deste problema seria utilizar as primitivas *sleep* (semelhante a uma operação suspend) e *wakeup* (semelhante a uma operação *resume*).

A solução dada é considerada parcial pois pode ocorrer que um sinal de *wakeup* seja enviado a um processo que não esteja logicamente *adormecido*, conduzindo os dois processos a um estado de suspensão que permanecerá indefinidamente, como indicado na sequência de etapas a seguir:

1. O *buffer* fica vazio.
2. O consumidor lê **contador**=0.
3. O escalonador interrompe o consumidor;
4. O produtor produz novo item e o coloca no *buffer*.
5. O produtor atualiza variável **contador** = 1.
6. O produtor envia sinal *wakeup* para consumidor pois **contador**=1.
7. O sinal de *wakeup* é perdido pois o consumidor não está logicamente inativo.
8. O consumidor é ativado, continuando execução, pois considera que **contador** =0.
9. O consumidor se coloca como inativo através de um *sleep.*

**c) Barreira:** Enquanto um processo está executando (ou tentando entrar) na região crítica, os estágios funcionam como barreiras (em cada barreira fica retido o último que lá chega): o estágio 1 deixa passar no máximo n-1 processos, o estágio 2 deixa passar no máximo n-2 e o último deixa passar apenas 1 processo. Isto garante a exclusão mútua. Para se convencer que os outros dois requisitos também são satisfeitos basta observar que um processo se bloqueia somente se algum outro está na sua frente;

como um processo não fica indefinidamente na sua região crítica, isto garante que todos os processos possam avançar.

d) **Monitor:** O monitor é quem gerencia o acesso a região crítica, e não o processo em si.

e) **TSL:** É uma solução de hardware para o problema da exclusão mútua em ambiente com vários processadores.

16. **Caracterize o funcionamento de "Escalonamento de Processos" com uso de prioridade de processos.**

    R: O sistema de escalonamento de processos com uso de prioridade acontece quando se utiliza uma variedade de processos interativos e em lote, estes processos são classificados conforme sua prioridade, e cada classe tem sua própria fila de prontos. O Round Robin é utilizado em todos os processos de uma mesma prioridade. Para evitar o *problema de inanição* (quando alguns processos nunca ganham a vez), pode-se definir períodos de tempo máximos para cada categoria: por exemplo, 70% para prioridade 1,20% para prioridade2, etc.

17. **Qual o melhor escalonamento para sistemas iterativos?**

R: O melhor escalonador de processos em sistemas interativos deve atender as seguintes exigências:

• Ser justo: todos os processos devem ser tratados igualmente, tendo possibilidades idênticas de uso do processador, devendo ser evitado o adiamento indefinido.

• Maximizar a produtividade (throughput): procurar maximizar o número de tarefas processadas por unidade de tempo.

• Ser previsível: uma tarefa deveria ser sempre executada com aproximadamente o mesmo tempo e custo computacional.

• Minimizar o tempo de resposta para usuários interativos. • Maximizar o número possível de usuários interativos.

• Minimizar a sobrecarga (overhead): recursos não devem ser desperdiçados embora algum investimento em termos de recursos para o sistema pode permitir maior eficiência.

• Favorecer processos bem comportados: processos que tenham comportamento adequado poderiam receber um serviço melhor.

• Balancear o uso de recursos: o escalonador deve manter todos os recursos ocupados, ou seja, processos que usam recursos sub-utilizados deveriam ser favorecidos.

• Exibir degradação previsível e progressiva em situações de intensa carga de trabalho.

No entanto, não se pode definir com exatidão qual o melhor escalonador dado que é necessário saber que processos serão executados e em que sequência, se é necessário ou não possuir lista de prioridades. Assim para cada caso divergente cabe ao desenvolvedor utilizar o escalonador que mais se adequa as necessidades do sistema.

**18. Diferencie os escalonamentos de threads no espaço do processo e no espaço do kernel (núcleo).**

R: Em sistemas multiprogramados (multi-tarefa), a cada instante um ou mais processos podem estar no estado pronto, e.g.:

– Processos do sistema e de usuários

– Processos curtos ou cumpridos/eternos

– processos interativos e não-interativos(e.g. programas CPU-bound ou jobs em batch (*))

O Escalonador é a parte do núcleo responsável por gerenciar a fila de prontos, e escolher qual dos processos prontos vai ser o próximo a usar CPU (de acordo com as prioridades dos processos) Também é responsável por ajustar (aumentar/diminuir) a prioridade dos processos;

**19. Como os sistemas operacionais tipo Unix fazem a manipulação dos sinais de processos?**

R: Os sinais são meios usados para que os processos possam se comunicar e para que o sistema possa interferir em seu funcionamento. Por exemplo, se o usuário executar o comando kill para interromper um processo, isso será feito por meio de um sinal.

Quando um processo recebe um determinado sinal e conta com instruções sobre o que fazer com ele, tal ação é colocada em prática. Se não houver instruções pré-programadas, o próprio Linux pode executar a ação de acordo com suas rotinas.

Entre os sinais existentes, tem-se os seguintes exemplos:

**STOP -** esse sinal tem a função de interromper a execução de um processo e só reativá-lo após o recebimento do sinal CONT;
**CONT -** esse sinal tem a função de instruir a execução de um processo após este ter sido interrompido;
**SEGV -** esse sinal informa erros de endereços de memória;
**TERM -** esse sinal tem a função de terminar completamente o processo, ou seja, este deixa de existir após a finalização;
**ILL -** esse sinal informa erros de instrução ilegal, por exemplo, quando ocorre divisão por zero;
**KILL -** esse sinal tem a função de "matar" um processo e é usado em momentos de criticidade.

**20. Como é possível a passagem de parâmetros da linha de comando à um processo.**

R: Em linguagem C podemos passar argumentos através da linha de comando para um programa quando ele inicia.

A função main recebe parâmetros passados via linha de comando como vemos a seguir:

**int main( intargc, char *argv[ ] )**

Onde:

**argc** – é um valor inteiro que indica a quantidade de argumentos que foram passados ao chamar o programa.

**argv** – é um vetor de char que contém os argumentos, um para cada string passada na linha de comando.

**argv[0]** armazena o nome do programa que foi chamado no prompt, sendo assim, argc é pelo menos igual a 1, pois no mínimo existirá um argumento.

Os argumentos passados por linha de comando devem ser separados por um espaço ou tabulação.

21. **Como o sistema Linux implementa o controle de semáforos? Relacione as funções e um código de exemplo.**

R: **Semáforos**

• Linux: duas implementações POSIX e SystemV

– POSIX: utilizado em threads

– SystemV: processos e threads. Complicado

Alocação

• intsemget(key_t key, intnsems, intsemflg);

– key: identificador do semáforo. IPC_PRIVATE cria um identificador único

– nsems: número de semáforos no conjunto

– Semflg

• IPC_CREAT:cria um novo segmento quando, um valor key é passado

• IPC_EXCL: exclusive, usado em conjunto com IPC_CREAT, caso o segmento já exista, retorna erro

• MODE FLAGS: flags de acesso 9 bits (TODOS/GRUPO/DONO). Utilizar constantes predefinidas (ver man2stat). Ex: S_IRUSR S_IWUSR : leitura e escrita para o dono da memória compartilhada 5 Inicialização

• intsemctl(intsemid, intsemnum, intcmd, union semunarg);

– semid= identificador retornado pelo semget

– semnum= número do semáforo

– Cmd

- GETALL
- GETNCNT
- GETPID
- GETVAL
- GETZCNT
- SETALL
- SETVAL

– /* arg for semctl system calls. */

unionsemun { intval; /* value for SETVAL */

structsemid_ds *buf; /* buffer for IPC_STAT & IPC_SET */

ushort *array; /* array for GETALL & SETALL */

structseminfo *__buf; /* buffer for IPC_INFO */

void *__pad; };

Operaçõessobresemáforos

• **intsemop**(**int**semid, **structsembuf** *sops, **unsigned**nsops);

– Semid= identificador retornado pela operação semget

– structsembuf {

ushortsem_num; /* semaphore index in array */

shortsem_op; /* semaphore operation */

shortsem_flg; /* operation flags */

};

sem_op= -1 (wait), 1 (signal), 0 dorme até o valor do semáforo for 0

sem_flg= IPC_NOWAIT = falha se o semáforo não está disponível

– nsops= número de operações

**CÓDIGO DE EXEMPLO:**

```
#define SHM_SIZE 4096

#include <stdio.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include<sys/sem.h>

#include<sys/types.h>

#include <sys/ipc.h>

unionsemun{

        intval; structsemid_ds *buf;        unsigned short *array; structseminfo
*__buf;

};

int main(){

        intshmid;

        intsemid;

        char* saddr;

        unionsemunarg;

        structsembuf wait={0, -1, 0};

        structsembuf signal={0, 1, 0};

        // Allocate

        shmid= shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT| S_IRUSR |
S_IWUSR);

        semid= semget(IPC_PRIVATE, 1, IPC_CREAT |S_IRUSR| S_IWUSR);

        saddr= (char *)shmat(shmid, 0, 0);

        // Inicializa arg.val= 1;

        semctl(semid, 0, SETVAL, arg);
```

**22. Como o sistema Linux implementa o controle de memória compartilhada? Relacione as funções e um código de exemplo.**

R: O sistema de memória compartilhada do Linux utilizam o modelo de memória UNIX.

- Processo aloca (allocate) um segmento no qual deseja compartilhar

- Processos que desejam acessar o segmento alocado, devem se vincular (attach) ao segmento

- Após a utilização da memória compartilhada, os processos devem se desvincular (detach)

- Quando a memória compartilhada não estiver mais em uso, a mesma deve ser liberada (deallocate)

Para Alocação (allocate)

shmget(key_t key, int size, intshmflg);

– Key: valor do identificador utilizado para designar a memória compartilhada. IPC_PRIVATE: utiliza um valor key único

– Size: tamanho da memória, arredondado para cima de acordo com o tamanho da página

– Shmflg: flags para controlar a criação da memória compartilhada

• IPC_CREAT:cria um novo segmento quando, um valor key é passado

• IPC_EXCL: exclusive, usado em conjunto com IPC_CREAT, caso o segmento já exista, retorna erro

• MODE FLAGS: flags de acesso 9 bits (TODOS/GRUPO/DONO). Utilizar constantes predefinidas (ver man2stat). Ex: S_IRUSR S_IWUSR : leitura e escrita para o dono da memória compartilhada

Para Vincular (attach)

• shmat(intshmid, const void *shm_addr, intshmflg)

– shmid: identificador retornado durante a alocação

– shm_addr: endereço no espaço local do processo, se NULL o SO escolhe um endereço livre

– shm_flg

• SHM_RND: arredonda para o próximo endereço múltiplo do tamanho de página

• SHM_RDONLY: somente leitura

Desvincular (detach) e Liberar (deallocate)

• Desvincular

– intshmd(const void *shmaddr);

• Liberar

– shmctl(intshmid, intcmd, structshmid_ds *bf);

• Verificar memórias compartilhadas

– ipcs

**Exemplo**

```
#define SHM_SIZE 4096

#include

 #include

#include

int main(){

        intshmid;

        char* saddr;

        // Allocate

        shmid= shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT| S_IRUSR | S_IWUSR);

        // Attach

        saddr= (char *)shmat(shmid, 0, 0);

        if (fork()==0){//children

                saddr[0]=65; sleep(5);

                printf("Filho %s", saddr);

                return 0;
```

```
        }
        saddr[1]=66; sleep(5);
        printf(" Pai %s" , saddr);
        return 0;
        wait();
        shmctl(shmid, IPC_RMID, 0);
}
```

**23. Considerando o código do Minix, explique o funcionamento das seguintes funções, correlacione com o código do MKERN0.c e também XV6.**

R: **do_clocktick()** – Essa função é chamada quando muitos processos necessitam ser realizados.

**pick_prock()** – Esta função decide que processo irá entrar em execução. Um novbo processo é selecionado configurando proc_ptr. Quando um novo usuário (ou inativo)processo é selecionado, este é registrado em bill_ptr. Assim a tarefa do clock pode dizer quem será cobrado pelo sistema.