# Sistemas Operacionais 2

**1. O que é um Sistema Operacional? Quais são suas funções?**

É um programa ou um conjunto de programas cuja função é servir de interface entre um computador e o usuário. O sistema operacional é uma coleção de programas que:

- inicializa o hardware do computador
- fornece rotinas básicas para controle de dispositivos
- fornece gerência, escalonamento e interação de tarefas
- mantém a integridade de sistema

**2. Descreva o modo de operação de um computador de acordo com a arquitetura de Von Neumann.**

A operação de um computador de acordo com a arquitetura de Von Neumann segue a seguinte sequência:

- Busca instrução; (uso do cache e mecanismos de prefetching, branch prediction, ... )
- Incrementa ponteiro de instruções; (Incremento equivalente ao tamanho da palavra)
- Decodifica e executa instrução. (Lógica no nível de micro-programa)
- Verifica interrupção (assíncrona (externa), trap, instrução (int))

**3. De que maneira são representados os processos em um sistema operacional?**

Um processo é composto por código, espaço em memória, contador de programa e seção de dados.

**4. O que é contexto de um processo?**

Denomina-se contexto de um processo o conjunto de informações que definem o estado do processo no momento que ele for interrompido. São as áreas correspondentes ao processo tais como espaço de endereçamento do usuário (texto, dados, pilha do usuário, memória partilhada), informação de controle (área-u; proc; pilha do kernel; mapa de tradução de endereços), credenciais (UID, GID), variáveis de ambiente e contexto hardware (PC, SP, PSW - processor status word, mmem regs, fpu regs).

**5. Como ocorrem as trocas de contexto?**

Todos os registradores referentes ao estado atual do processo são salvos na piha de execução do processo pelo SO para que ele possa ser restaurado no mesmo ponto de execução, e outro processo

da fila de prontos é carregado para ser executado. Para tanto, o sistema operacional executa as seguintes etapas:

1. Processo efetua chamada de sistema bloqueante;
2. SO recebe interrupção de entrada/saída;
3. SO empilha contador de programa, PSW registradores, etc;
4. SO invoca rotina de tratamento de interrupção;
5. Rotina salva dados da pilha na tabela de processos (TP) e atualiza estado do processo para bloqueado;
6. Rotina invoca o escalonador de processos que retorna o novo processo a ser executado (dentre os processos prontos);
7. Rotina carrega PSW, registradores, etc para o novo processo (obtidos na TP);
8. Rotina atualiza contador de programa para o valor obtido do novo processo;
9. A partir deste instante, a próxima instrução já pertence ao novo processo, finalizando a troca de contexto.

**6. Descreva estratégias usadas por sistemas operacionais para o compartilhamento do tempo de execução de um processador entre diversos processos.**

Para realizar o compartilhamento de tempo de execução entre diversos programas é possível sobrepor entrada e saída de dados com instruções (multiprogramação) ou fornecer a cada processo uma fatia de tempo no qual ele pode executar sendo que mediante o término dessa, o processo será interrompido.

**7. Descreva técnicas usadas pelo sistema operacional para interagir com os controladores de dispositivos.**

- DMA: (DMA tem a ver com SO?)
- Driver de dispositivo: cada controlador de dispositivo possui um driver a ele associado por meio do qual o controlador de dispositivo se comunica com o SO. Para iniciar uma operação de E/S, o driver carrega os registradores apropriados para dentro do controlador. Este, por sua vez, examina o conteúdo desses registradores para determinar que ação deve ser realizada. Quando a ação estiver concluída o controlador informa ao driver via uma interrupção que terminou a operação. Após isso o driver retorna o controle ao SO e os dados pertinentes a ação.

**8. O que são interrupções? Descreva os tipos existentes e os eventos aos quais estão associados.**

Uma interrupção é uma requisição para que o processador interrompa o que está fazendo e passe a realizar operações sobre outros dados. Funciona através de um sinal que avisa o processador para mudar o fluxo de execução, indo para uma rotina de tratamento da interrupção, onde são salvos todos

os dados do estado corrente do processo atual e novos dados são inseridos nos registradores e no Registrador de Instrução. Após o fim do processamento, os dados do processo antigo são recuperados e o fluxo de execução volta para onde estava.

- Externas: sinal vindo do hardware para avisar que uma E/S está terminada.
- Internas: (traps) é uma interrupção gerada por software, causada por um erro ou por uma requisição específica de um programa do usuário, para que um serviço do SO seja relacionado.

**9. Como ocorre o tratamento de uma interrupção? Quais ações são desempenhadas pelo processador e quais são realizadas pelo Sistema Operacional?**

Quando ocorre uma interrupção o estado do processo é salvo e a execução é desviada para a rotina do tratador de interrupção, em um endereço fixo na memória. O tratador de interrupção determina a causa, efetua a restauração do estado e executa uma instrução de retorno para se voltar ao estado de execução anterior a interrupção.

- Processador:
    - termina a execução da instrução em andamento;
    - sinaliza o reconhecimento da interrupção
    - salva PSW e PC
    - carrega novo valor de PC (baseado na interrupção)
- SO:
    - salva o restante do contexto do processo
    - processa interrupção
    - restaura o contexto
    - restaura PSW e PC

**10. Como deve ser uma rotina de tratamento de interrupção no sistema operacional? Que ações devem ser executadas?**

**Idem 09 (?)**

**11. Considerando o gerenciamento do processador, o que é concorrência?**

É a capacidade do SO de manter vários processos sendo executados, um por vez, realizando troca e contexto entre eles num tempo definido de acordo com uma estratégia de escalonamento. É a base para um sistema multitarefa.

**12. O que é paralelismo de execução?**

É o modelo onde a execução do software é dividida em partes relativamente independentes para aproveitamente mais eficiente dos recursos ociosos do hardware

**13. O que são chamadas de sistema? Como são implementadas e de que maneira um programa de usuário tem acesso aos serviços do sistema operacional?**

São uma interface entre os programas de usuário e o SO para que possam ser requisitados serviços do núcleo (instruções privilegiadas que não podem ser executadas pelos programas de usuário). Essas SystemCalls possuem funções com o mesmo nome em C e podem ser chamadas passando atributos ou retornando valores, e através delas são executadas instruções em modo kernel.

**14. Descreva os modelos de programação baseados em memória compartilhada e em memória distribuída**.

Para memória compartilhada:

- Threads: processos são separados em sequências paralelas, que compartilham variáveis dentro do programa
- Programas sequências com diretivas de programação para especificar paralelismo: uso de biblioteca para paralelização automatizada

Para memória distribuída:

- Clusters: passagem de mensagens entre os processos em máquinas distintas
- Grid: ativação remota do software para processamento de dados em vários computadores, possuindo um controle central que ativa a execução de determinado software nas máquinas locais. As máquinas estão em locais distintos.
- Nuvem (cloud): computação distribuída à recursos virtualizados, acessíveis pela Internet.

**15. Em um sistema operacional, o que é um programa shell? Como funciona?**

Shell é um programa que permite ao usuário interagir com o SO através de comandos em modo texto. Os comandos digitados são passados para o kernel para serem executados

**16. Num shell, o que significa execução em foreground e execução em background? De que maneira o shell trata essas duas formas de execução de processos?**

Se o programa realiza sua execução em foreground, a shell fica esperando até o fim de sua execução para permitir que o usuário digite mais comandos. Já na execução em background, a shell continua a permitir a execução de outros comandos enquanto o programa está sendo executado.

O shell trata as duas maneiras igualmente. Só que quando o processo está em background e ele solicita leitura ou escrita do shell, ele fica bloqueado, até que ele volte para foreground e o processo de leitura/escrita possa acontecer.

**17. Num sistema unix, como são identificados os processos?**

Cada processo possui um identificador único, chamado PID. Devia a natureza da criação dos processos, eles possuem um PPID também, que é o PID associado ao pai do processo.

**18. Num shell de ambiente unix, para que serve o comando ps?**

O comando *ps* lista os processos iniciados a partir do shell corrente.

**19. Num sistema unix, o que são e para que servem os sinais? Dê exemplos de sinais.**

O mecanismo padrão do UNIX para informar a um processo de que um evento ocorreu é o sinal.SIGCHLD, SIGKILL, SIGSTOP.

**20. Num sistema unix, para que serve o comando kill? Como são identificados os processos destinatários do sinais?**

O comando kill serve para enviar um sinal a um processo, identificando-o através do seu PID.

**21. Considerando a definição de chamadas de sistema no padrão POSIX, explique de que forma as chamadas aos serviços do sistema operacional são implementadas. Como é o**

**código inserido nos programas de usuário pelas chamadas de sistema e como os acessos aos serviços do SO são realizados em tempo de execução (runtime)?**

A chamada do sistema coloca os valores apropriados nos registradores do hardware e na pilha do processo, chamando em seguida a instrução de interrupção. A partir dos valores dos registradores e da pilha, o sistema saberá qual syscall executar.

**Para as questões a seguir, considere as APIs POSIX.**

**22. Para que serve e como funciona a chamada fork( )?**

A chamada fork() serve para criar um processo filho a partir da "duplicação" do processo que a chamou. Ela retorna o *pid* do filho e 0, quando vista pelo processo pai e processo filho, respectivamente.

- O filho não herda: memory locks, semaphore adjustments, record locks, timers, oustanding asynchronus IO operations e asynchoronus IO contexts.
- O filho tem um *pid* único;
- *ppid* do filho = *pid* do pai;
- Contadores de tempo de CPU e utlização de recursos do processo são setados para 0;
- Conjunto de sinais pendentes do filho: vazio;

**23. Diferencie as chamadas fork( ) e vfork( ).**

*pid_t vfork(void);*
*pid_t fork(void);*

A função vfork() tem o mesmo efeito que a função fork() exceto que o comportamento é indefinido se o processo criado por vfork() ou modifica algum dado além da variável de tipo *pid_t* usado para armazenar o valor de retorno de vfork(), ou retorna da função na qual vfork() foi chamada, ou chama qualquer função antes de chamar _exit() ou uma função da família exec().

**24. Para que serve a chamada system( )? Ela promove o paralelismo ou a concorrência de execução?**

*int system(const char *command);*

A chamanda system() serve para executar um comando especificado em *command* (chamando /bin/sh -c *command*) e retorna a execução do processo que a chamou após ter executado *command.* (concorrência de execução)

**25. Descreva a operação das chamadas exec( ). O que acontece com o contexto do processo que as executa? Quais informações do processo são preservadas após as chamadas?**

*int execl(const char *path, const char *arg, ...);*
*int execlp(const char *file, const char *arg, ...);*
*int execle(const char *path, const char *arg, ..., char * const envp[]);*
*int execv(const char *path, char *const argv[]);*
*int execvp(const char *file, char *const argv[]);*

Parâmetros do exec( ):
e - An array of pointers to environment arguments is explicitly passed to the child process.
l - Command-line arguments are passed individually to the function.
p - Uses the path environment variable to find the file named in the path argument to be executed.
v - Command-line arguments are passed to the function as an array of pointers.

As funções da família exec() substituem a imagem do processo que as chamou pela imagem do processo a ser chamado (definido no primeiro parâmetro das funções exec). O contexto do processo chamador é perdido, porém algumas informações do processo são mantidas, como: pid, ppid, priority, owning user e owning group.

**26. Descreva as funções wait( ) e waitpid( ). No que são diferentes?**

*pid_t wait(int *status);*
*pid_t waitpid(pid_t pid, int *status, int options);*

As funções wait() e waitpid() são usada para suspender a execução do processo pai até que aconteça uma mudança de estado (terminar, ser parado por um sinal ou continuado por um sinal) em um de seus filhos e obter informações sobre o filho cujo estado mudou.
A chamada wait() suspende a execução do processo chamador até que qualquer um dos filhos termine. Já a chamada waitpid() suspende a execução até que um filho especificado pelo argumento *pid* mude de estado. Por default a chamada waitpid() somente espera por filhos terminados, mas esse comportamento pode ser modificado via o argumento *options.*

**27. O que são sinais para processos? Quais são os sinais definidos pelo padrão POSIX e a quais eventos estão associados? Que ações são definidas por padrão para o tratamento dos sinais?**

Sinais são os meios utilizados para que os processos possam se comunicar e para que o sistema possa interferir em seu funcionamento.
As ações associadas por padrão para o tratamento de sinais são: Term, Ign, Core, Stop, Cont.

- SIGHUP: ‘’hang up’’ detectado no terminal controlador ou morte do processo controlador;
- SIGINT: interrupção do teclado;
- SIGQUIT: saída do teclado;
- SIGILL: instrução ilegal;
- SIGABRT: sinal de abortar vindo de *abort()*;
- SIGFPE: exceção de ponto flutuante;
- SIGKILL: envio do sinal *kill*;
- SIGSEGV: referência inválida de memória;
- SIGPIPE: *pipe* quebrado;
- SIGALRM: sinal de timer vindo de *alarm()*;
- SIGTERM: envio do sinal de término;
- SIGUSR1: sinal definido pelo usuário;
- SIGUSR2:sinal definido pelo usuário;

- SIGCHLD: filho parado ou terminado;
- SIGCONT: continuar se parado;
- SIGSTOP: parar o processo;
- SITTSTP: stop digitado em tty;
- SITTTIN:entrada tty para um processo em background;
- SITTTOUT:saída tty para um processo em background

Existem outros, mas chega né?

**28. De que maneira o sistema operacional oferece o tratamento de sinais para os processos?**
O sistema operacional oferece o tratamento de sinais das seguintes formas:

- ignorar o sinal: nenhuma ação é tomada;
- capturar e tratar o sinal: o kernel irá suspender a atual sequência de execução do processo e irá pular para uma função previamente registrada. Uma vez executada a função o processo irá voltar para o ponto que estava antes do sinal ser capturado.
- usar a ação default: todo sinal tem uma ação default a ele associada que deve ser tomada no caso de nenhum dos dois casos acima se aplicarem.

**29. Descreva as funções para manipulação de sinais:**

- *sighandler_t signal(int signum, sighandler_t handler):* chama o *handler* (função definida pelo usuário, SIG_IGN, SIG_DFL) definido previamente para o sinal representado por *signum.*
- *int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact):* Altera a ação (*act*) tomada por um processo no recebimento de um sinal específico (*signum*);
- *int sigpending(sigset_t *set):* armazena em *set* o conjunto de sinais que estão com sua entrega pendente;
- *int sigprocmask(int how, const sigset_t *set, sigset_t *oldset):* usada para alterar a máscara de sinais baseada em *how (SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK).* Se *oldset* for não nulo o valor antigo da máscara é armazenado nele. Se *set* for nulo a máscara não é alterada.

**30. Como é enviado um sinal para um processo?**
Cada processo tem associado a ele quatro campos de bits (bitfields): PENDING, BLOCKED, IGNORE e CAUGHT. O sinal é enviado para um processo através da chamada do sistema kill(), que marca o bit correspondente no PENDING do processo alvo. Quando um processo é selecionado pelo scheduler, antes de ganhar o processador, o kernel checa se há sinais pendentes para ele. Caso haja, são checados os campos BLOCKED e IGNORE. Se o sinal está marcado para ser bloqueado, o kernel o mantém em PENDING. Caso o sinal esteja marcado como IGNORE, o kernel descarta o sinal. Se o sinal não estiver nem bloqueado nem ignorado, é executado o handler relativo a este sinal e marcado em CAUGHT.

**31. Em que momento uma função associada ao tratamento de um sinal por um processo é**

**executada?**
Uma função associada ao tratamento de um sinal por um processo é executada quando esse processo captura um sinal que teve para ele definido, como tratamento, a execução dessa função.

**32. Quais são as políticas de escalonamento definidas no Linux?**
FIFO, RR, BATCH e OTHER

**33. Descreva as políticas de escalonamento a seguir:**

- **SCHED_FIFO:** um processo executando sobre a política FIFO irá continuar executando até que um processo de maior prioridade requisite execução, seja bloqueado ou chame sched_yield().
- **SCHED_RR:** é similar ao FIFO, porém adiciona uma condição para processos de mesma prioridade (fatias de tempo)
- **SCHED_OTHER:** representa a política padrão. Todos os processos executando sobre essa política tem prioridade estática 0. Consequentemente serão preemptados por processo usando RR ou FIFO. O escalonador usa o valor nic para priorizar os processos dentro dessa política.
- **SCHED_BATCH:** processos nessa classe só irão executar quando não existirem outros processo executáveis no sistema, mesmo que os outros processos tenham exaurido suas fatias de tempo.

**34. Descreva as funções a seguir, relacionadas com o controle do escalonamento de um processo:**

- *int getpriority(int which, int who):* retorna a maior prioridade (o menor valor de *nice*) de qualquer um dos processos especificados. *Which* deve ser: PRIO_PROCESS, PRIO_PGRP ou PRIO_USER; bem como *who* deve especificar um *pid*, *pgid* ou *uid* respectivamente
- *int setpriority(int which, int who, int prio):* definite a prioridade de todos os processos especificados para *prio.*
- *int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)*: define a política de escalonamento do processo representado por *pid* para *policy.* Qualquer parâmetro é setado via *param.*
- *int sched_getscheduler(pid_t pid):* retorna a política de escalonamento do processo representado por *pid.*
- *int sched_setparam(pid_t pid, const struct sched_param *param):* define parâmetros de escalonamento associados ao *pid.*
- *int sched_getparam(pid_t pid, struct sched_param *param):* retorna via *param* os parâmetros de escalonamento associados a *pid.*
- *int sched_get_priority_max(int policy):* retorna o valor máximo válido de prioridade para *policy.*
- *int sched_get_priority_min(int policy):* retorna o valor mínimo válido de prioridade para *policy*.
- *int sched_rr_get_interval(pid_t pid, struct timespec *tp):* retorna em *tp* a duração da fatia de tempo alocada para *pid*.

**35. Para que serve a função sched_yield( )? Dê exemplo de uma situação de uso.**
*int sched_yield(void);*
A função sched_yield() faz com que a thread chamadora libere a CPU. A thread é então movida para o

final da lista referente a sua prioridade estática e uma nova thread é executada.
Escalonamento.
http://kerneltrap.org/Linux/Using_sched_yield_Improperly (achei a discussão interessante).

**36. Em sistemas com múltiplos processadores (ou núcleos), para que serve o valor CPU affinity mask, associado pelo sistema operacional a um processo (ou thread)?**
Ele se refere à probabilidade de um processo (ou thread) ser escalonado consistentemente no mesmo processador (ou núcleo).

- *int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)*: define a affinity mask associada a *pid* como *mask.*
- *int sched_getaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)*: armazena a affinity mask de *pid* em *mask.*
- *void CPU_CLR(int cpu, cpu_set_t *set)*: desatrela *cpu* do processo.
- *int CPU_ISSET(int cpu, cpu_set_t *set)*: verifica se um dado *cpu* no sistema está atrelado ou não ao processo.
- *void CPU_SET(int cpu, cpu_set_t *set)*: atrela *cpu* ao processo.
- *void CPU_ZERO(cpu_set_t *set)*: zera todos os bits em *set.*

**37. Como são definidos os limites associados à execução de processos? Qual é a implicação de um limite ser atingido?**
Os limites associados à execução de processos são definidos pelo seu *soft limit* (valor definitod pelo kernel) e *hard limit* (teto para o *soft limit*). Os recursos já possuem valores padrões para ambos os limites, porém eles podem ser alterados atráves da função *setrlimit* (*soft limit* varia de 0 até seu *hard limit* e apenas processos priviletgiados podem aumentar seu *hard limit* [a ação de abaixá-lo é irreversível]). Um limite ao ser atingido faz com que seja enviado um sinal para o processo para avisá-lo disso e bloqueia as operações relativas ao recurso exaurido.

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
struct rlimit {
rlim_t rlim_cur; // Soft limit
rlim_t rlim_max; // Hard limit (ceiling for rlim_cur)
};
```

**38. Defina  recursos que podem ser limitados para cada processo:**
*RLIMIT_AS:* tamanho máximo da memória virtual do processo em bytes.
*RLIMIT_CORE:* tamanho máximo de core file que esse processo pode criar;
*RLIMIT_CPU:* limite de tempo de CPU em segundos;
*RLIMIT_DATA:* tamanho máximo do semento de dados do processo;
*RLIMIT_FSIZE:* número máximo de arquivos que o processo prode criar;
*RLIMIT_LOCKS:* limite no número combinado de flock() e fcntl()
*RLIMIT_MEMLOCK:* número máximo de bytes da memória virtual que podem ser bloqueado;
*RLIMIT_MSGQUEUE:* número máximo de bytes que um usuário pode alocar para message queues POSIX;
*RLIMIT_NICE:* máximo valor até o qual um processo pode diminuir seu valor de *nice;*
*RLIMIT_NOFILE:* specifica um valor um maior que o número máximo de descritores de

arquivos que podem ser abertos;
*RLIMIT_NPROC:* número máximo de processos que podem ser criados para o *real user ID;*
*RLIMIT_RSS:* limite (em páginas) do conjunto residente do processo
*RLIMIT_RTPRIO:* máximo level de prioridade de tempo real que um processo sem CAP_SYS_NICE pode requisitar
*RLIMIT_SIGPENDING:* número máximo de sinais que podem estar em espera para o usuário;
*RLIMIT_STACK:* tamanho máximo da pilha do processo, em bytes;

**40. De que maneira são contabilizados pelo SO os recursos consumidos pelos processos? Como é o acesso a essas informações e quais informações são providas pelo Sistema Operacional sobre a execução de um processo?**

Os recursos consumidos pelos processos são contabilizados atráves da estrutura *rusage*. E o acesso a essas informações é feito através de *getrusage* na qual se define *who* (RUSAGE_SELF ou RUSAGE_CHILDREN), assim como se passa uma estrutura rusage para poder acessar as informações providas. Essas sendo: ru_utime (user time used), ru_stime (system time used), ru_maxrss (maximum resident set size), ru_ixrss (integral shared memory size), ru_idrss (integral unshared data size), ru_isrss (integral unshared stack size), ru_minflt (page reclaims), ru_majflt (page faults), ru_nswap (swaps), ru_inblock (block input operations), ru_oublock (block output operations), ru_msgsnd (messages sent), ru_msgrcv (messages received), ru_nsignals (signals received), ru_nvcsw (voluntary context switches), ru_nivcsw (involuntary context switches)

```
int getrusage(int who, struct rusage *usage);
struct rusage {
struct timeval ru_utime; /* user time used */
struct timeval ru_stime; /* system time used */
long   ru_maxrss; /* maximum resident set size */
long   ru_ixrss; /* integral shared memory size */
long   ru_idrss; /* integral unshared data size */
long   ru_isrss; /* integral unshared stack size */
long   ru_minflt; /* page reclaims */
long   ru_majflt; /* page faults */
long   ru_nswap; /* swaps */
long   ru_inblock; /* block input operations */
long   ru_oublock; /* block output operations */
long   ru_msgsnd; /* messages sent */
long   ru_msgrcv; /* messages received */
long   ru_nsignals; /* signals received */
long   ru_nvcsw; /* voluntary context switches */
```

*long ru_nivcsw; /* involuntary context switches */*
*};*

**41. De que maneira pode ser determinado o tempo decorrido (elapsed time) na execução de partes do código de uma aplicação? Exemplifique.**
Pode-se determinar o tempo decorrido na execução de partes do código ao armazenar o tempo antes dessa parte ser iniciada e o tempo ao final fazendo, por fim, uma subtração do final menos inicial. Podemos usar por exemplo a função *getrusage* que salva no seu segundo parâmetro uma estrutura *rusage* e utilizar os campos relacionados ao tempo (*ru_utime, ru_stime*).

**42. Apresente um exemplo de código que ilustra uma forma de contabilizar o tempo de CPU gasto para a execução de um trecho de um programa.**

```
struct rusage myusage1, myusage2;
struct timeval t1, t2, elapsed;
getrusage(0, &myusage1);
/* ... */ /* código cujo tempo se quer medir */
getrusage(0, &myusage2);
timeradd(&myusage1.ru_utime, &myusage1.ru_stime, &t1); /* aqui somam-se
myusage1.ru_utime e myusage1.ru_stime */
timeradd(&myusage2.ru_utime, &myusage2.ru_stime, &t2); /* aqui somam-se
myusage2.ru_utime e myusage2.ru_stime */
timersub(&t2, &t1, &elapsed); /* t2 - t1 */
```

**43. O que são pipes? Como são mantidos pelo sistema operacional e como funcionam?**
Pipes são canais de dado unidirecionais que podem ser usados para IPC. Ao criar um pipe define-se suas duas pontas: pipefd[0] (read end), pipefd[1](write end). Quando um processo quiser escrever algo ele deve usar a função *write* passando como o descritor o *pipefd[1].* Já se quiser ler de um pipe deve passar o *pipefd[0].*

**44. Qual é a relação entre pipes e named pipes (fifos)?**
Os FIFOs são por vezes chamados named pipes e podem ser utilizados para estabelecer canais de comunicação entre processos não relacionados ao contrário dos API do SO UNIX pipes, que exigem sempre um ascendente comum entre os processos que ligam

**45. Como é feito o redirecionamento dos canais básicos de entrada e saída (stdin e stdout) usando pipes?**
Processo pai cria 2 pipes, in e out, e faz um fork-exec. Processo filho fecha (close()) a ponta de escrita de in e a ponta de leitura de out e redireciona, por exemplo, seu stdin para a ponta de leitura de in (com dup2()). Processo pai fecha a ponta de escrita de out e a ponta de leitura de in.

*int dup2(int oldfd, int newfd);*
*FILE *popen(const char *command, const char *type);*
*int pclose(FILE *stream);*

**46. O que é IPC? Descreva os principais mecanismos.**

Inter Process Communication.

- Pipe: são canais de dado unidirecionais
- FIFO: permitem o acesso ao pipe a partir da identificação no sistema de arquivos.
- Fila de mensagens: lista ligada de mensagens mantidas pelo kernel
- Memória compartilhada: é um espaço de memória compartilhado por processos que permite a comunicação sem o intermédio do kernel.
- Semáforos: mecanismo de sincronização de processos que podem ser de 2 tipos: binário ou de contagem.
-

**47. Para que serve e como funciona o mecanismo de filas de mensagens da API system V?**
Filas de mensagens são utilizadas para comunicação e sincronização (no caso de espera por uma mensagem específica). Inicialmente uma fila é criada usando o *msgget (key, msgflg)* (ou retorna uma fila já existente pelo campo *key*). Após a criação mensagens são enviadas e recebidas utilizando as funções *msgsend* e *msgrcv* respectivamente.

**48. Como são identificadas as mensagens no envio e no recebimento?**
As mensagens são identificadas através o *mtype* que pode ser obtido pelo parâmetro *msgp* (que é uma estrutura *msgbuf* contendo o tipo da mensagem e o seu conteúdo).

```
int msgget(key_t key, int msgflg);
struct msgbuf {
long mtype; // message type: deve ser >0 no envio
char mtext[1]; // message data: como é último campo, permite envio de msgs maiores
};
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

**49. Para que serve e como funciona o mecanismo de filas de mensagens da API POSIX?**
O kernel mantém as mensagens salvas em memória, podendo elas ter mais de uma pagina de alocação. Em seguida é criado um vetor de ponteiros ordenado por prioridade, no qual cada posição armazena um ponteiro para a respectiva mensagem. Serve para a troca de mensagens com prioridade entre processos e threads no mesmo sistema.

**50. Como são identificadas as mensagens no envio e no recebimento?**
No padrão POSIX as mensagens são identificadas pela sua prioridade, a mensagem com maior prioridade será a primeira a sair da fila.

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
mqd_t mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio,
const struct timespec *abs_timeout);
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
```

*ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio, const struct timespec *abs_timeout);*

**51. Para que servem e como funcionam os semáforos definidos pela API System V?**
Servem para sincronizar dois processos. Ele é criado através do *semget* quando *key* é IPC_PRIVATE ou se não existe um semáforo associado a essa *key* e IPC_CREAT está definido nas *semflag.* Os semáforos são manipulados através do *semctl* e do *semop.*

**52. Como implementar chamadas não bloqueantes para esses semáforos?**
Setando a *ssbuf.flg* com IPC_NOWAIT.

**53. Usando operações não bloqueantes, esquematize o código da aplicação para que não haja erro no uso do recurso protegido pelo semáforo.**

*int semget(key_t key, int nsems, int semflg);*
*int semctl(int semid, int semnum, int cmd, ...);*
*int semop(int semid, struct sembuf *sops, unsigned nsops);*
*int semtimedop(int semid, struct sembuf *sops, unsigned nsops, struct timespec *timeout);*

**54. Considerando os semáforos providos pela API System V, como é ajustado o valor inicial do semáforo?**
Inicializa a variável *val* da estrutura do semáforo com o valor desejado. (sem_un.val = 1 e semctl(semid,0,SETVAL,sem_un))

**55. Também para os semáforos SystemV, esquematize a implementação das funções lógicas sem_wait( ) e sem_signal( ), descrevendo os parâmetros necessários e suas operações internas.**

```
int lock(int semid, int num_sem){
    struct sembuf ssbuf;
    ssbuf.sem_num=num_sem;
    ssbuf.sem_op=-1;
    ssbuf.sem_flg=1;    // flag == 1 indica acesso bloqueante
    return(semop(semid,&ssbuf,1)!=-1);}

intunlock(int semid, int num_sem){
    struct sembuf ssbuf;
    ssbuf.sem_num=num_sem;
    ssbuf.sem_op=1;
    ssbuf.sem_flg=1;    // flag == 1 indica acesso bloqueante
    return(semop(semid,&ssbuf,1)!=-1);}
```

**56. De que maneira processos distintos, sem um ancestral comum (não gerados através de fork() de um mesmo "pai") podem vir a ter acesso a um mesmo semáforo?**
pshared (POSIX)

**57. Para que servem e como funcionam os semáforos definidos pela API POSIX:**
Servem para a sincronização entre processos e threads.

- *int* ***sem_init(****sem_t *sem, int pshared, unsigned int value):* inicia o semáforo com um valor especificado, *pshared* indica se o semáforo será visível por threads de outros processos (entre processos).
- *int* ***sem_trywait****(sem_t *sem):* permite realizar uma operação não bloqueante, retornando EAGAIN se não for bem sucedida.
- *int* ***sem_post****(sem_t *sem):*incrementa o semáforo indicado, se o valor torna-se maior que 0, processo ou thread ali bloqueado é acordado
- *int* ***sem_wait****(sem_t *sem):* decrementa contador associado ao semáforo, se valor for maior que 0, decremento é realizado e processo prossegue. Caso contrário, chamada é bloqueada até que seja possível realizar operação, ou até o recebimento de um sinal.
- *int* ***sem_destroy****(sem_t *sem):* destrói o semáforo

**58. Como implementar chamadas não bloqueantes para esses semáforos?**
*sem_trywait* permite realizar uma operação não bloqueante, retornando EAGAIN se não for bem sucedida.

**59. Usando operações não bloqueantes, esquematize o código de uma aplicação para que não haja erro no uso do recurso protegido pelo semáforo.**

**60. É possível compartilhar esses semáforos entre processos distintos? (pesquisar named semaphores)**

**61. Como são definidas e usadas áreas de memórias compartilhadas entre processos com a API System V?**
shmget cria um bloco de memória compartilhada ou retorna um identificador de um bloco já existente. Tamanho é especificado como parâmetro. Para que possa ser usado, contudo, ponteiro deve ser associado ao bloco alocado, o que é feito com a chamada shmat. Já shmctl realiza operações de controle sobre a área alocada, com a sua liberação depois de todas as desassociações de ponteiros. IPC_STAT, IPC_SET, IPC_RMID, IPC_INFO, SHM_STAT, SHM_LOCK, SHM_UNLOCK**.**

**62. Esquematize o ajuste de uma área compartilhada e compare com o uso de ponteiros comuns.**

**63. De que maneira processos distintos podem vir a ter acesso a uma área de memória compartilhada usando a API System V?**
*shmat* com mesma *shmid* para os processos a serem compartilhados.

*int shmget(key_t key, size_t size, int shmflg);*

*void *shmat(int shmid, const void *shmaddr, int shmflg);*
*int shmdt(const void *shmaddr);*
*int shmctl(int shmid, int cmd, struct shmid_ds *buf);*

**64. Faça uma comparação entre os diversos mecanismos disponíveis para comunicação e sincronização entre processos executados num mesmo computador (baseados em memória compartilhada). Para cada mecanismo, discuta: localização dos dados (user space x kernel space) e cópias em memória necessárias, estruturas de dados que precisam ser mantidas pelo Sistema Operacional, atrasos, acesso ao sistema de arquivos (se for relevante), etc.**