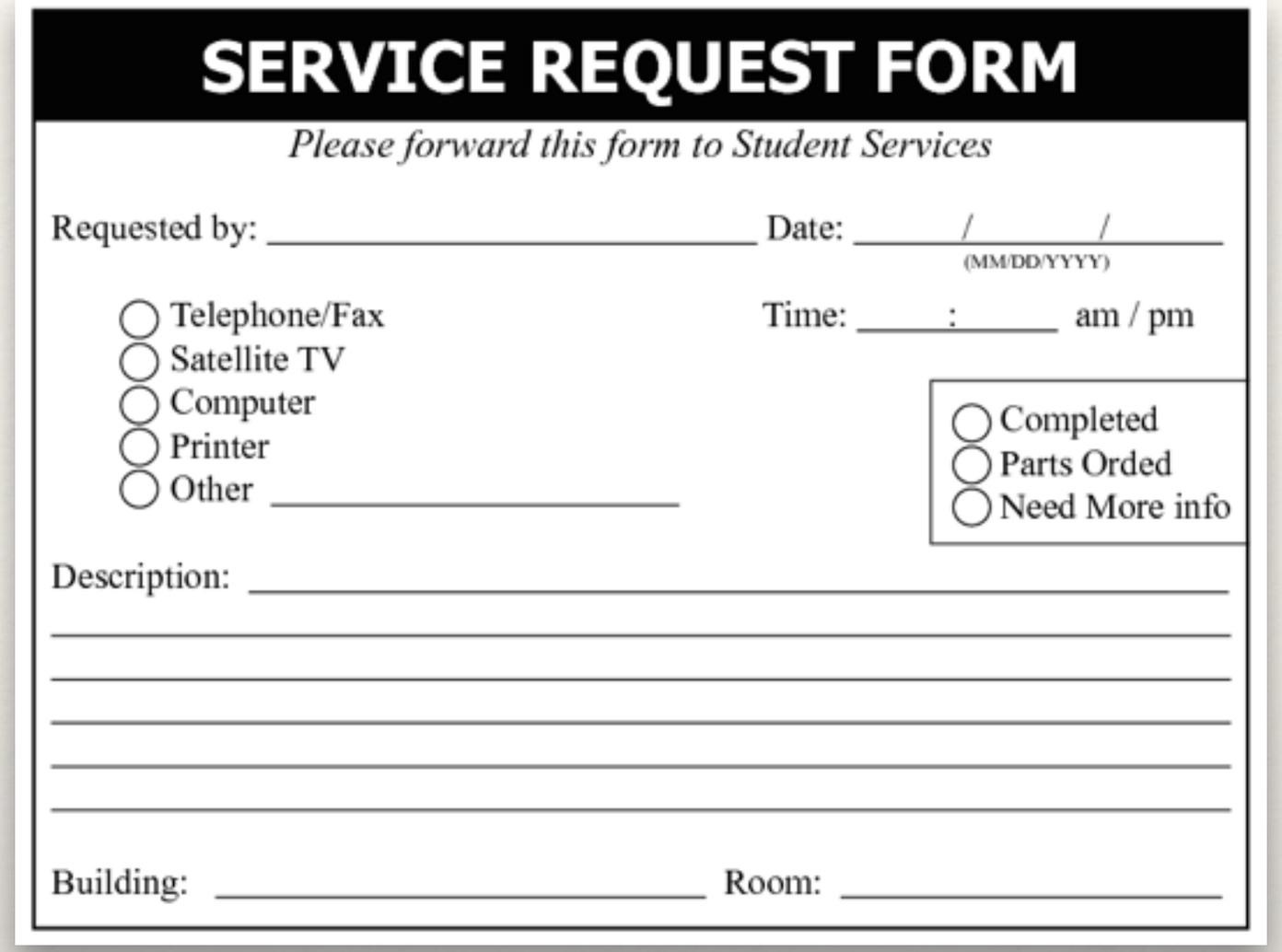
Prof. Ricardo Inácio Álvares e Silva

Principais Conceitos

Sistemas Operacionais



SERVICE REQUEST FORM

Please forward this form to Student Services

Requested by: _____ Date: ____/____/____
(MM/DD/YYYY)

◯ Telephone/Fax
◯ Satellite TV
◯ Computer
◯ Printer
◯ Other _____

Time: ____:____ am / pm

◯ Completed
◯ Parts Orded
◯ Need More info

Description: _____

_____

_____

_____

_____

_____

Building: _____ Room: _____

Chamadas ao Sistema

Exemplos de chamadas ao sistema

Recapitulando

- Quais as funções de Sistemas Operacionais?
  - Gerenciar recursos escassos do sistema computacional
  - Abstrair a funcionalidade específica do hardware
- Para tal, um Sistema Operacional deve oferecer funções comuns a vários sistemas (*de hardware*), independente das características específicas e dispositivos presentes
  - Essas funções são como a biblioteca padrão do Java
  - São conhecidas como **chamadas ao sistema** (*system calls*)
- Chamadas ao sistema são específicas para um sistema operacional
  - O conceito e funcionalidades são praticamente as mesmas, só mudam nomes e maneiras de chamada

Padrão POSIX

- Padroniza cerca de 100 chamadas ao sistema
  - Um certo programa que utilize somente chamadas POSIX compila e funciona corretamente em todos os sistemas POSIX, sem modificações
  - Normalmente, sistemas operacionais UNIX são POSIX
  - Windows não é UNIX mas possui um módulo tradutor de chamadas POSIX
  - Um sistema implementa chamadas de sistema no padrão POSIX não fica impedido de implementar chamadas extras, de acordo com as necessidades do sistema

Recapitulando o funcionamento

- Já estudamos o funcionamento de chamadas ao sistema na aula de *Revisão de Hardware*
- As chamadas ao sistema nada mais são que **rotinas** executadas pelo processador
- Porém, envolvem instruções e acessos **protegidos**, acessíveis apena com a CPU em modo **superusuário**
- Se um programa precisa da funcionalidade de uma rotina dessas, ele faz uma chamada ao sistema

- Passos para realização da chama ao sistema:

    1. O programa arranja os parâmetros nos registradores e posições de memória pertinentes à chamada ao sistema desejada
    2. O programa indica qual serviço deseja, em um registrador, padronizado pelo SO, para isso
    3. O programa realiza uma interrupção - há uma instrução especialmente para isso
    4. O sistema operacional recebe o controle do processador e verifica o por que daquela interrupção, lendo o registrador de serviços
    5. O sistema decide se vai atender ou não o pedido. Em caso positivo, ele chama a subrotina desejada
    6. A subrotina faz o seu trabalho e, se houver, escreve os valores de retorno em posição de memória acessível ao programa que a requisitou
    7. O sistema operacional devolve o controle para o programa

- No MIPS, a instrução de interrupção é:
  - *syscall*
    - não possui nenhum parâmetro direto na instrução
    - Cada SO utiliza sua própria forma para descobrir porque foi feito aquele *syscall* pelo programa do usuário
- No Sistema Operacional do MARS
  - *$v0* é usado para indicar qual serviço o usuário deseja
  - cada serviço utiliza alguns registradores para receber parâmetros (consultar manual)

```
.data
CARTEIRA:
    .word 300
STR_SALARIO:
    .asciiz "O salario calculado foi de: "

.text
    lw $a0, CARTEIRA
    li $a1, 12              # Meses de cálculo
    li $a2, 1000            # Valor do salário
    li $a3, 437             # Imposto mensal bruto
    jal COMPUTA_SALARIO # Chama a rotina do salario
    sw $v0, CARTEIRA

    li $v0, 4               # Nº serviço escrita de string
    la $a0, STR_SALARIO # Endereço da frase a escrever
    syscall

    li $v0, 1               # Serviço de escrita de inteiros
    lw $a0, CARTEIRA
    syscall

    li $v0, 10              # Serviço de fim de programa
    syscall

# Nossa rotina de salários
COMPUTA_SALARIO:
    add $a0, $a0, $a2   # Soma salario com carteira
    sub $a0, $a0, $a3   # Subtrai os impostos
    addi $a1, $a1, -1   # decrementa um mês
    bne $a1, $zero, COMPUTA_SALARIO
    add $v0, $a0, $zero # Copia o resultado para retorno
    jr $ra              # Retorna após o "jal"
```

Exemplos de chamadas POSIX

- Considere a seguinte chamada ao sistema

```
count = read(fd, buffer, nbytes);
```

- Funcionalidade:
  - lê uma quantidade `nbytes` de bytes
  - do arquivo apontado por `fd`
  - armazena em `buffer`
  - `count` recebe a quantidade real de bytes lidos, ou -1 em caso de erro.

```
pid = fork()
```

- Funcionalidade:
  - cria um processo filho, clone do processo atual
  - após a chamada haverá dois processos independentes
    - instâncias do mesmo programa
- retorna:
  - 0 para o processo-filho
  - PID do processo-filho para o processo pai

```
pid = waitpid(pid, &statloc, options)
```

- Funcionamento:
  - O processo atual fica suspenso enquanto o processo `pid` não terminar execução
  - Se `pid` for -1, o sistema espera qualquer um processo filho existente terminar
  - Ao final, `statloc` indica valor de retorno, entre 0 a 255
  - O retorno da função indica qual `pid` foi esperado

`s = execve(name, argv, environp)`

- Funcionamento:
  - Substitui o programa do processo atual pelo programa indicado por `name`
  - Os argumentos de execução do programa são indicados por `argv`
  - Há variações dessa função, com nomes similares, e funcionalidade levemente modificada. O conjunto de todas essas funções é normalmente referenciado apenas como `exec()`

`exit(status)`

- Funcionamento:
  - Termina a vida do processo
  - `status` indica um valor de retorno entre 0 a 255, que é utilizado por `statloc` de `waitpid()`

- Implementação mais simples de um terminal de linha de comando

```
#define TRUE 1

while (TRUE) {
    type_prompt();
    read_command(command, parameters);

    if (fork() != 0) {
        /* Significa que eh o processo pai */
        waitpid(-1, &status, 0);
    } else {
        /* Significa que eh o processo filho */
        execve(command, parameters, 0);
    }
}
```

Outros exemplos rápidos

Variadas

Chamada	Descrição
`s = chdir(dirname)`	Muda o diretório de trabalho
`s = chmod(name, mode)`	Muda as proteções binárias de um arquivo
`s = kill(pid, signal)`	Envia um sinal a um processo
`seconds = time(&seconds)`	Retorna o tempo passado desde 1º/1/1970

Gerenciamento de arquivos

Chamada	Descrição
`fd = open(file, how, ...)`	Abre um arquivo para leitura e/ou escrita
`s = close(fd)`	Fecha um arquivo aberto
`n = read(fd, buffer, nbytes)`	Lê dados de um arquivo para o buffer
`n = write(fd, buffer, nbytes)`	Escreve dados de um buffer para um arquivo
`position = lseek(fd, offset, whence)`	Move o apontador de arquivo
`s = stat(name, &buf)`	Recebe informações sobre o estado do arquivo

Gerenciamento de arquivos e diretórios

Chamada	Descrição
`s = mkdir(name, mode)`	Cria um novo diretório
`s = rmdir(name)`	Apaga um diretório vazio
`s = link(name1, name2)`	Cria um arquivo-ligação
`s = unlink(name)`	Remove um arquivo-ligação
`s = mount(special, name, flag)`	Monta um sistema de arquivos
`s = umount(special)`	Desmonta um sistema de arquivos

Chamadas ao sistema do Windows

- Assim como o POSIX padroniza chamadas ao sistema para vários UNIX, **Win32 API** padroniza as de Windows
- Diferentes no paradigma de programação:
  - POSIX: self-service, o processo se utiliza do S.O.
  - Win32: event driven, o S.O. aciona o processo
- Possui dezenas de milhares de funções
- A cada nova versão do Windows, Win32 ganha novas chamadas
  - a implementação pode modificar, mas mantém compatibilidade
- Gerencia vários serviços, como montagem de janelas

Exemplos de chamadas da Win32

Gerenciamento de arquivos e diretórios

Chamada	Descrição
`CreateProcess`	Cria um novo processo
`WaitForSingleObject`	Pode esperar o término de um processo
`ExitProcess`	Termina o processo
`CreateFile`	Abre ou cria um novo arquivo
`CloseHandle`	Fecha um arquivo
`ReadFile`	Lê dados de um arquivo
`WriteFile`	Escreve dados a um arquivo
`SetFilePointer`	Move o apontador de um arquivo
`GetFileAttributesEx`	Retorna variados atributos de um arquivo
`CreateDirectory`	Cria um novo diretório
`RemoveDirectory`	Remove um diretório vazio
`DeleteFile`	Apaga um arquivo existente
`SetCurrentDirectory`	Muda o atual diretório de trabalho
`GetLocalTime`	Retorna o tempo atual