

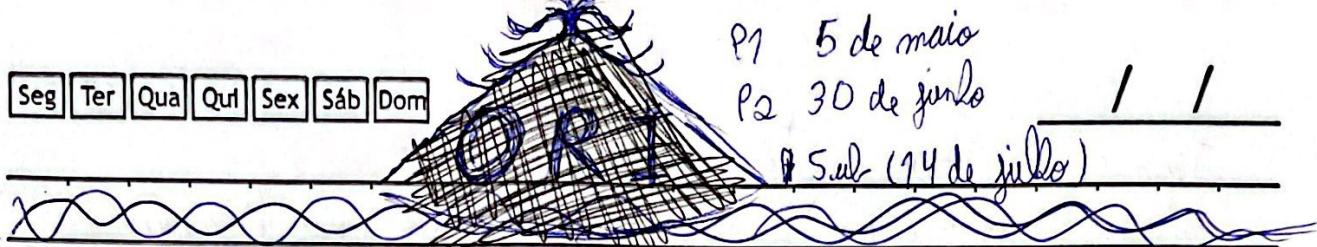
Seg Ter Qua Qui Sex Sáb Dom

P1 5 de maio

P2 30 de junho

/ /

5.ub (14 de julho)



$$MF = MT \cdot 0,65 + MP \cdot 0,35$$

- Um Sistema Computacional consiste em:

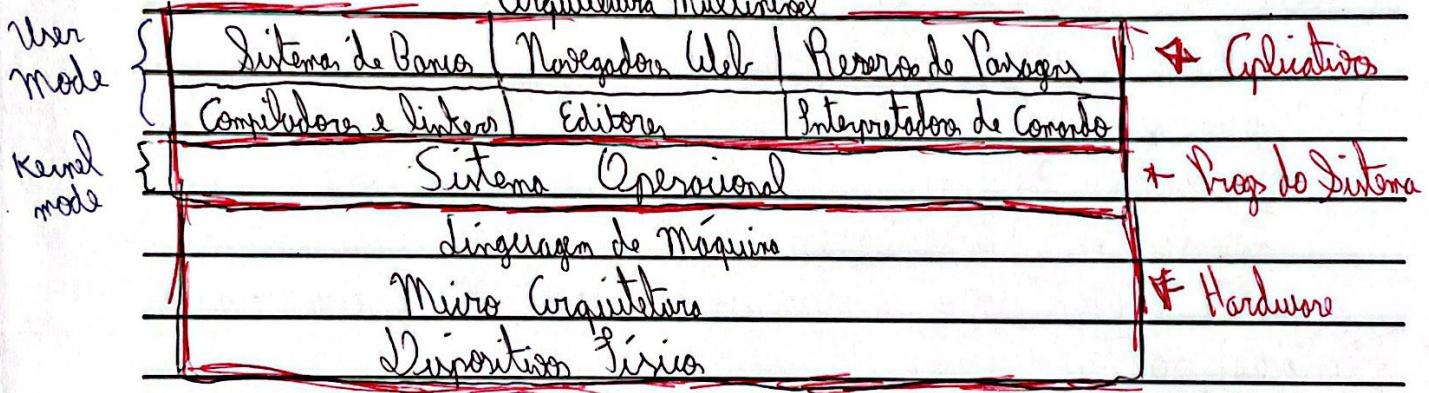
1 ou mais processadores

memória

Dispositivo de entrada e saída.

→ O objetivo do SO é gerenciar recursos e fazer a ponte entre a Aplicativo / rotina de E/S ↔ Hardware

Arquitetura Multicamada



- Interações com o usuário:

- Linguagem de comando JC (Job Control Language)
- Interação Gráfica
- Interação com Programas
- System Calls (Chamadas de sistema)

UNIX Description

• fork	create a new process	{	exec	move the file pointer
• waitpid	wait a process to exit	{	stat	get various of file attribute
• execve	create process = fork + execve	{	mkdir	create a new directory
• exit	terminate execution	{	rmdir	remove an empty directory
• open	create a file or open a file	{	rm	remove a file
• close	close file	{	cd	change the current working dir
• read	read data from file	{	chmod	modify an permission de arquivo?
• write	write data to a file	{	Kill	Kill a process? FORONI

Formas de Processamento do SO

- Serial (Monoprograma) = recursos dedicados a um único programa.
- Concorrente (Multiprograma) = "diferentemente dedicada entre vários prog em diferentes etapas"

Introdução ao Sistema Operacional (texto)

↳ Shell = programa de interface que o usuário manda texto

↳ GUI = " " " " " " " gráficos (Graphic User Interface)

- SO é um software executado no modo Kernel e que realiza 2 funções principais:
 - ¹ Fornece recursos para programas e aplicativos (ao invés de um hardware complexo)
 - ² Faz o gerenciamento de recursos de hardware.

1- Ex: como a arquitetura de um computador é complicada → criam-se os drivers → arquivos
 ↳ O trabalho do SO é criar laços abstratos

2- 2 prog querem imprimir / usar memória / ... é o SO que aloca cada um na sua vez.

↳ gerenciamento de recursos: Multiplexação (compartilhamento)

• Tempo: ex: 1 CPU 3 programas: 1º prog usa 4 ms, 2º prog usa 4 ms, 3º usa 3 ms, ...

• Espaço: ex: Memória: é melhor dividir ela (RAM) entre os prog do que dar tudo a um só vez

História dos SO's

1º computador digital = Charles Babbage (1792-1871) → não funcionou → 1ª programadora Ada Lovelace.

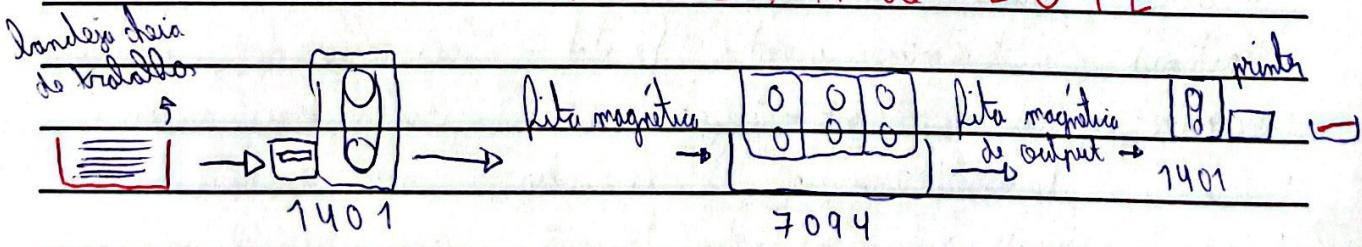
• 1ª Geração (1945-55): Valvulas (vacuum tubes) → Alan Turing

↳ 2ª Geração → era programada na mão, conectando fios e tubos → fazia cálculos simples.

• 2ª Geração (55-65): Transistors e sistemas em lote (batch)

computadores eram chamados de mainframes, caríssimos, o programador escrevia o prog em FORTRAN ou Assembly, perfurava cartões e entregava os lotes de cartões ao operador.

SISTEMA de LOTE



Seg Ter Qua Qui Sex Sáb Dom

Estrutura de um Cartão \$JOB

\$JOB 10,777 - Conta TODAR → Título, nome do cara

\$FORTRAN → correção compilador FORTRAN

\$LOAD → correção prog. digitado recém compilado

\$RUN → roda o programa → \$END → fim do trabalho "\$JOB"

• eram comandos aos SO's, primitiva das shells modernas

• 3^a Geração (65-80) - CI e Multiprogramação

↳ IBM 360 foi a 1^a grande linha a usar CI's (Circuito Integrado)

↳ queria agendar executar e ^{concorrência (diversos)} mas era difícil fazer um PC assim → ^{ultra} PUG's

Mas, popularizou técnica de Sistemas Operacionais como a Multiplexação:
antes a CPU ficava caída esperando I/O, perdendo muito tempo que
poderia estar produzindo, no comando o tempo de espera de I/O era puro 90%.

Assim, partilhou-se a memória entre os JOBS, enquanto um
estava esperando I/O, o outro poderia ser executado.



- Também adicionaram o spool (simultaneous peripheral operations online), isto é, salvavam o conteúdo dos fitas diretamente da disco, assim quando um JOB adiante, era só carregar o próximo na posição, não precisava esperar por outro
- TimeSharing: c/ 20 pessoas online 17 usavam, a CPU pode fazer trocas rápidas a esses 3 e em segundo plano trabalhar em grandes JOB's em batch.

• 4^a Geração (80 - agora): Computadores Pessoais.

↳ desenvolvimento do chip LSI (large scale integration) → contendo milhões de Transistores

↳ x86 → processador baseado na arquitetura de 1970, com instruções que concorrem com 8086. x86-32 (32 bits) → x86-64 (64 bits)

• 4^a Geração (90 - agora)

1.3 Resumo de Hardware de Computadores

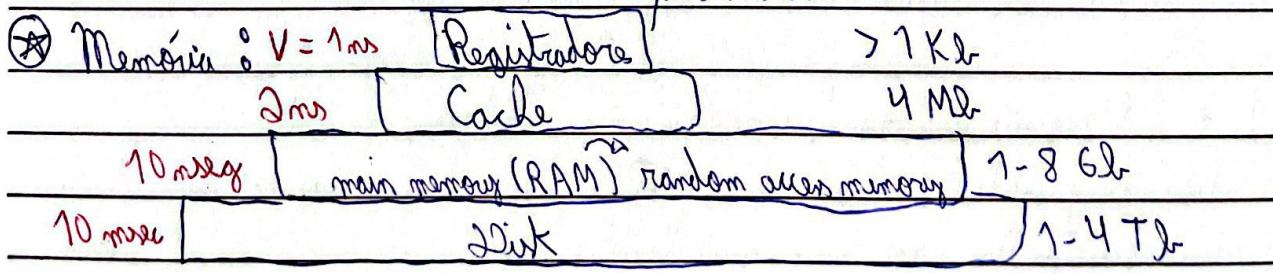
No computador moderno, a CPU, memória, dispositivo de E/S, ... estão conectados a um barramento "bus"

- Processadores:** recebe do PC, busca instruções na memória e as executa.
- Cada processador pode executar determinadas instruções. Um X86 não executa instruções de um ARM.
 - Algumas instruções da memória demoram mais tempo do que executá-las, por isso elas usam os registradores.
 - PC program counter: contém o endereço de memória da próxima instrução a ser lida.
 - SP (ponteiro de pilha): aponta para o topo da pilha de procedimentos entrados mas não terminados.
 - Frame: contém os dados desse quadro (é o quadro) mantém esses dados em registros.
 - PSW (Program Status Word): contém vários bits de controle, prioridade de CPU, modo Kernel ou usuário...
O SO para programar em execução constantemente quando está realizando a multiplexação da CPU; ele deve salvar todos os registros para serem restaurados quando o programa for executado posteriormente.
 - Pipeline: executar instruções ao mesmo tempo (Cpq.)
 - CPU Superescalares: várias instruções ao mesmo tempo; um holding buffer segura os resultados, quando vê que já tem para fazer a instrução de escrita, gera instruções feitas fora de ordem.

Programas de usuário funcionam em modo de usuário; não pode fazer algumas coisas tipo memória protegida, também não pode entrar em modo Kernel
 alarmado do sistema trap muda de usuário para Kernel → transfere o comando do SO -

Outros traps executados pelo hardware para iniciar uma exceção especial X
 → origem do processo base → baralhar da memória

- Multithreading: modo de thread se for uma que demora muito tempo, executa isso bem rápido, fazendo com que pareça que os nós de 2 CPU's existam 4 se eles estiverem cheios, só o SO pode acionar alternando 2 threads no mesmo CPU enquanto o outro está dormindo.
- multicore: processa ~~processa~~ multicore tem 4 minicores dentro deles cada um com seu próprio 32x32 bits ou 64x64 bits



- Cache = dividido em linhas de cache normalmente com 64 bytes e endereço 0-6³
Quando um prog precisa de uma palavra, o endereço de cache procura se a linha desejada está no cache, se estiver é um "cache hit". Caso contrário o conteúdo de memória é levado a RAM.

↳ É usado para melhorar o desempenho ex: "/Home/Izquierdo/Doc/CAII/oi.c" armazenado no cache esse endereço faz com que esteja sempre (armazena a conserva).

Cache L1 = menor (Kb) e não demora tanto para dar a resposta

↳ L2 = maior (Mb) demora 2 ciclos de clock Intel (compatível L2) e AMD tem um L2 para cada núcleo.

• RAM = grande e volátil (não guarda o PC desliga)

• ROM = Read only memory → gravado na fábrica (dados para iniciar o PC)

↳ EEPROM e Flash → não é volátil e pode ser gravado + demora + para escrever.

• Flexos: ↳ também salvam BIOS

Almoxarifado →  (trabalho) → 512 bytes

• Dispositivos de E/S: São controlados que facilitam o trabalho do SO, entao envia só um comando e já responde. → interface para o controlador = driver.

a entrada • Busy wait: O prog de usuário emite uma chamada de sistema para saída podem fazer um ET, o Kernel trabalha em um procedimento de loop enquanto o dispositivo estiver pronto, entao quando estiver pronto para o SO que retorna ao prog.

• método 2: das interrupções

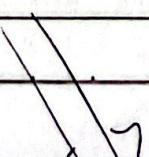
• .. 3: chip DMA Direct Memory Access nem pergunta só interrupção, por isso a CPU tem um dispositivo de interrupções

★ Barramentos:

PCI e (Peripheral Component Interconnect Express) ↳ 32 bits

PCI (regular)  é compartilhado e tem que garantir que neguem juntas

(conexões de SOs)



FORONI

- **Processos** = Basicamente um programa em execução
 - ↳ Um container que guarda toda as informações para a execução do programa. (seu endereço de prog.exe, dados e sua pilha)
 - ↳ As informações sobre o processo ficam salvas na **tabella de processos** (process table), para que possam ser retornadas de onde parou depois
 - ↳ Um signal para libertar: → um timer acabou, se foi feito algo indevido, ...
 - ↳ Tem UID (User Identification) para o usuário que fez o processo ou (GID group).
- **Espacos de endereços**: deve-se gerenciar e proteger a memória principal para que um processo não interfira em outro ou no SO; já que é sólido os dados dos processos. Se um deles tiver mais espaço de endereço que a memória ~~sistêmica~~ principal, é usado **memória virtual**. (Ou seja, guarda parte da memória do processo na principal, parte no disco, e dispositivos entre eles... indo para frente e para trás entre eles.)
- **Copistas**: arquivos são separados em diretórios, se forem arquivos de um dispositivo externo é acoplado à árvore de arquivos e pode ser acessado da mesma forma.

'/dev'

- 2 tipos de arquivos especiais: de Bloco (block files) e arquivos especiais de caracteres
- 1- existem para的模样 'arquivos que não entram diretamente em Bloco', como discos
- 2- usado para模样 'dispositivos que saem ou produzem um fluxo de caracteres' (impressoras, ...)
- ↳ Pipes (tubos)
- ↳ Processo A → Pipe → Processo B
- Ex: /dev/lp'

→ não conseguem entre processos que se arremelham muito a arquivos, se A quer mandar algo para B deixa no pipe e o B lê (como um arquivo).

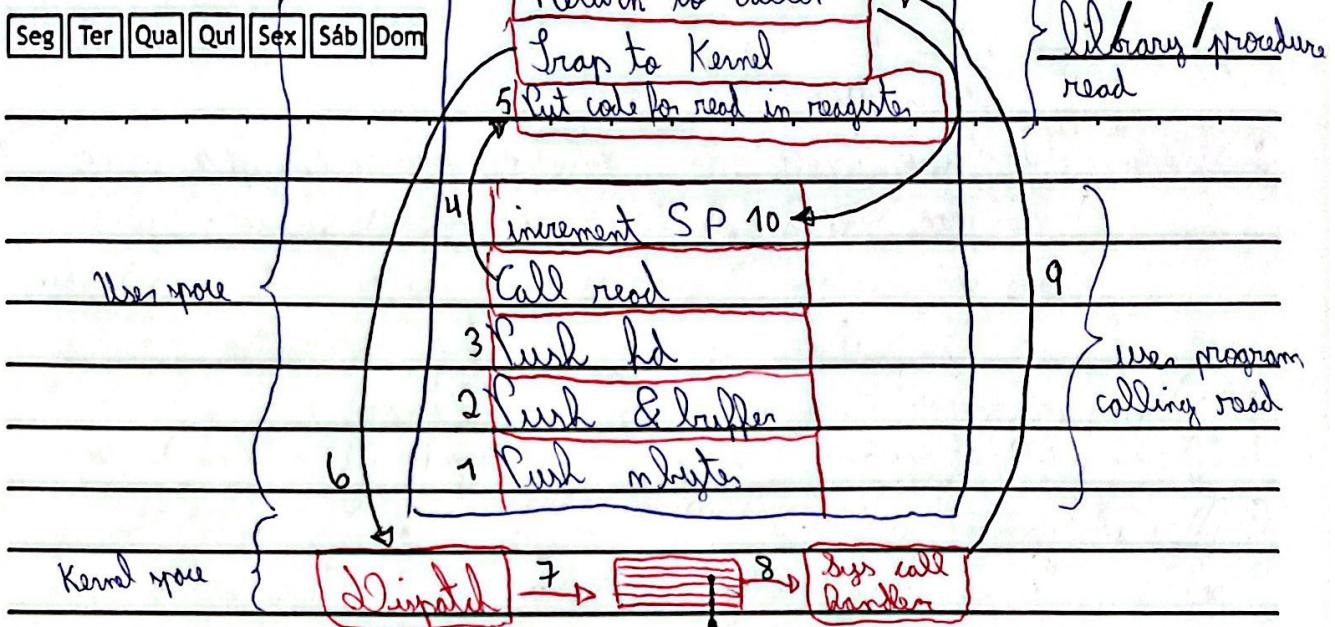
- **Dispositivos de E/S**: não necessários, já que precisam de drivers.
- **Proteção de Arquivos**: 9 bits código, 1 para o proprietário e resto os outros grupos R/W/X-
- **Shell**: local me: ? 0 0
- **Chamadas de Sistema**: quando um prog. de usuário precisa de algo como ler um arquivo, ele passa o comando para o SO por meio da system call : trap, então o SO descobre o que o processo chamador deseja lendo os parâmetros. Ex: ler um arquivo

com:

read(fd, buffer, nbytes);
 where nº de bytes lidos, ou especifica o onde colocar nº de bytes a serem
 os dados lidos
 mem. de endear o arg antes

ERROS → se der erro, salva o erro no variável global "errno".

Seg Ter Qua Qui Sex Sáb Dom



Chamada de sistema ≠ Chamada de procedimento

muda para o Kernel mode → dir. o endereço relativo ou absoluto.

pela sua vez faz fixo ou tem

8 bits no instrução para explicar o endereço da instrução

• pid = fork () | cria um processo filho idêntico e retorna seu pid

• pid = waitpid (pid, &status, options) | wait for a pid to terminate

• s = execve (nome, args, environ) | Replace a process core image.

↓
nome do arquivo a ser executado

• Chamadas de sistema para diretórios : mkdir, rmdir, link →
cada diretório é um arquivo com vários pares (i numbers, nome ASCII).
i number é o número do arquivo e o nome é seu nome. assim o link cria
em dois arquivos o mesmo nome : ex: pra o i-node (16, nome-joso) e só (16, g-) →
normalmente em diretórios diferentes.

• Chamadas de ^{sistema} arquivo para arquivos : open, close, read, write, ioctl
ioctl muda a posição do ponteiro de leitura do arquivo.

continuando : mount ("dev/sd", "/mnt"); ↗ mnt ↗
↓ nome do dispositivo de bloco ↓ onde monta

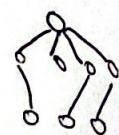
• Outras : chmod , Kill → maneira como os usuários e processos lidam com os;
se um processo estiver preparado para receber o sinal ele chama o signal handler,
se não ele mata o processo.

FORONI

Estrutura do SO

① **Sistemas Monolíticos**: Todo o SO é executado como único prog no modo Kernel

- 1 - um prog principal que inicia o procedimento de serviço solicitado
- 2 - um conjunto de procedimentos de serviço que realizam as reqst. calls
- 3 - " " " utilitários que ajudam no procedimento de serviços



② **Sistema com Camadas (Layered)**: Feito por Ilustra e em escala.

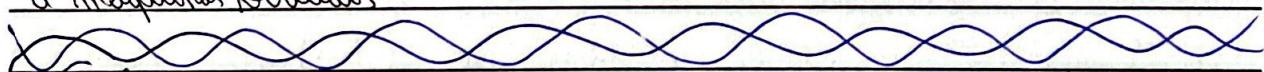
↳ Ilustra o SO, e Livro → o principal (camada mais fina) é a lógica de processos

③ **MicroKernels (µKernels)**: Nem todos os serviços do SO são no Kernel; isto é, em outros módulos do SO para evitar que bugs quebrem o sistema todo, rede/sem fios, monitores, ...
↳ Ficam na camada de usuário

④ **Modelo Cliente/Servidor**: distingue dois tipos de processos, os do cliente, que fazem requisições por serviços, e o do servidor, que faz o trabalho e envia de volta a resposta.

⑤ **Máquinas Virtuais**: são cópias exatas de hardware físico, modo Kernel, dispositivos de E/S, interruptões e tudo mais. JVM → Java Virtual Machine → rodar prog. Java em qualquer PC.

⑥ **Externares**: partilham uma máquina real a fim de distribuir recursos à máquinas virtuais.

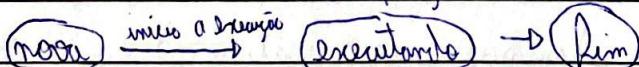


Cula 2 → Tipos e Estrutura do SO

① **Monoprograma ou Monotarefa**: 1 prog ativo que fica na RAM até seu fim

② **Multiprograma ou Multitarefa**: + de um prog na memória, compartilhamento da CPU, ...

↳ não usa todos os recursos ao mesmo tempo, deixa coisas ociosas, mas é simples.



2 → mantém vários programas em memória ao mesmo tempo, mais otimizado, mas complicado
↳ UNIX, Windows, ...

↳ 3 Tipos: Em lote, sistema de tempo compartilhado, tempo Real
(batch) Timelsharing → apreende o tempo virtual.

- Sistemas em Batch : lote de tarefas (conjunto de JOBS).

↳ não há interação com o usuário e o job durante a execução.

 → UCP (CPU)

- SO Multitarefa Interativo :

↳ permite interações c/ usuário no formato de diálogo, compartilhamento de tempo.

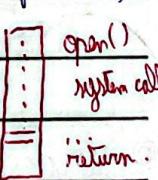
- Tipos de Estrutura do SO : Já scrito, página anterior.

~~Chamadas de sistema~~ { Cula 3 } : Chamadas de sistema

↳ se uma aplicação precisa fazer algo privilegiado, ela faz uma chamada alterando para o modo Kernel

↳ Elas são realizadas através de intrusões "traps" : interrompem o software, resolvendo e processando a demanda do sistema.

a) Aplicativo faz chamada ao sistema (Trap).

b) Gravação de um tabela, o SO determina o endereço da rotina. (→  return)

c) Rotina de Serviço é criada (rotina compatibilida)

d) Serviço solicitado é executado e o controle retorna ao aplicativo onde ele parou (término nos registros)

- Utilizam interface de chamadas de sistema para deixar mais fácil

para o usuário (Wrappers) Ex: Win32 (Windows), POSIX (UNIX), JAVA API (JVM)

Ex: em um código .c printf('Sólo');

↳ system call write() e exit()

↑ é do UNIX, expandindo a compatibilidade com

Interrupções : no nível de software → as traps (evento interno)

no nível de hardware (eventos externos) → sinal elétrico, E/S, ou disk.

O tratamento da interrupção é igual ao do trap.

Ex: um disco terminou a leitura, ele reporta isso ao controlador de interrupções que repassa isso a CPU, a CPU faz o que recebeu a interrupção e procura o processo que estava esperando por essa interrupção.

Questionário 1 - SO

P1?

$$15, 8, 17, 24, 28, 27$$

+ + + + +

tempo trap 2 do design T
média

m -?

G 9

$$\frac{\Delta S}{\Delta t} = \frac{\Delta V}{\Delta t} = 10 = 200 \text{ micro-s}$$

M 6

m -9

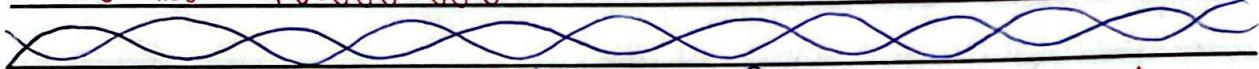
K 3

$$\Delta t = \frac{10}{200}.$$

p

8.000.000.000 bits

$$10 \text{ ms} = 10.000.000 \text{ ns}$$

SEMANA 2

"uma instância de um programa"

Processos: São atividades realizadas pelo computador, ex: pegar algo do disco, como isso demora, a CPU faz várias outras coisas enquanto espera. em exemplo

Possuem paralelismo. (Uma tarefa é organizada em vários CPU's).

+ O Modelo de Processo:

O processo não é organizado sequencialmente em uma CPU, que troca de processo enquanto ele espera por outro (chamado multiprogramação). O controlador de programa salva os endereços de cada tarefa em seus respectivos contadores lógicos. Em tarefas que estão relacionadas com o tempo é essencial que a CPU volte na hora certa, sem se demorar em um outro processo.

+ Criação de Processos: a criação de processos pode ocorrer pelas seguintes causas:

1 - Inicialização do sistema

2 - Execução de um comando de sistema de criação de processo por um usuário em exemplo

3 - Solicitação do usuário para criar um novo processo.

4 - Início de uma tarefa em Date.

Daemons: processos rodando em segundo plano que não têm interação com o usuário

No UNIX, a criação de um processo é feita através da system call: fork, que cria um processo filho e ele é que faz, por exemplo, o execve.

↳ tem a mesma imagem na memória e variáveis que o processo pai quando criado.

FORONI

Mas o processo pai e filho têm espacos de endereços distintos, de um ou em outro mundo uma palavra no seu espaço de endereço, o outro não sabe.

+ Terminos de processos: pode ocorrer por:

1- Saída normal (voluntário) → chamado exit (UNIX)

2- Erro fatal (involuntário) → ex: não ter o arquivo requisitado no programa.

3- Saída por erro (voluntário) → ex: fórmula inválida, executa instrução ilegal

4- Morte por outro processo (involuntário) → Kill

+ Hierarquia de Processos:

Um processo pai e seus filhos (e os filhos dos filhos) não são um grupo. Se um signal for enviado ao pai, ele é passado para todo o grupo, não um deixa o que é dentro do PC iniciar, o processo init é pego da memória; assim, c/ o signal para todos os outros não filhos dele, em uma árvore de processos.

(Windows não tem muita hierarquia e pode ter vários filhos, o UNIX tem e só pode).

+ Estados de Processos:

-
- 1- O processo é bloqueado aguardando uma entrada
2- O escalador seleciona outro processo (multiprogramação)
3- O escalador seleciona esse processo
4- A entrada torna-se disponível.

+ Implementação de Processos

↳ Para implementar o modelo de processo, o SO tem uma tabela de processos com entrada para cada processo. Nela tem info importante sobre o estado do processo: tipo seu controles de programa, ponteiro de pilha, ... ai ele pode parar e voltar de hora. Esta tabela tem os campos: gerenciamento de processo, gerenciamento de memória e gerenciamento de arquivos (com várias coisas em cada). Tratamento e escalonamento de interrupção: fig da pg. 8 (semana 2).

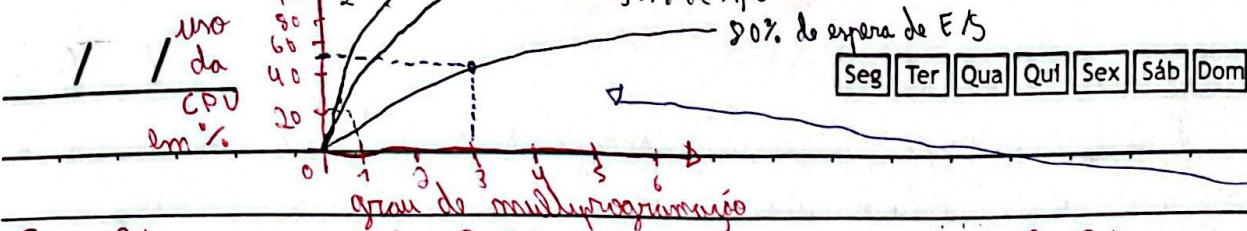
+ Modelando a Multiprogramação:

Se p é o fração de tempo que um processo para esperando que dispositivo de E/S estejam prontos e m é o número de processos.

$$\text{Utilização da CPU} = 1 - p^m$$

Grau de Multiprogramação em função de m :

FORONI



Ex: Uma memória de 8 GB; 2 GB para o SO e 2 GB para cada programa de usuário; se cada prog tem uma espera de 80% para entrada e saída, qual é a utilização da CPU (ignorando sobreuso do SO):

R: $1 - 0,8 = 0,2$ ou em torno de 49%. * estarem 3 progs simultaneamente → adiciona mais uma memória de 8 GB, ou seja, grau de multiprog = 3.

↳ grau de multiprogramação chega a $7^{\frac{1}{3}} \rightarrow 79\%$.

* gráfico bom na pg 9 da semana 2.

• Threads: são itens para dividir tarefas menores. Exemplo:
sendida com o Thread deputante e Thread operário
dormente enquanto é cheia de repisão dormente enquanto não lhe é dada tarefa
com os threads é possível ter paralelismo e... threads

Threads → Paralelismo, chamados de ^{de sistema} bloquantes

Processo monothreaded → Não paralelismo, chamados de internos bloquantes
Máximo de estados finitos → Paralelismo, chamados não bloquantes, interrupções

+ Modelo de thread clássico

O modelo de processo é baseado em: agrupamento de recursos e execução.

↳ Embora uma thread devo executar em algum processo, a thread e seu processo são conceitos diferentes, já que o processo é para agrupar recursos e a thread não entidade escalonada para execução na CPU.

Assim, ter múltiplos threads executando em um processo é como ter múltiplos processos executando em paralelo no processador. → do processo (trabalhando juntas)

Threads compartilham espaço de endereçamento e outros recursos, já os processos compartilham a memória física e ^{impressoras} outras... threads = processos dentro multithreading = múltiplos threads no mesmo processo.

Mas, apesar de compartilharem variáveis globais, endereço de memória, etc, cada thread possui sua própria pilha, contendo uma estrutura para cada rotina chamada ^{retorno} mas não ~~rotina~~, variáveis locais e endereço de **FORONI** retorno para quando a rotina for encerrada.

Como o Multithreading normalmente começa com um único thread que cria outro com thread-create especificando a rotina que esse thread executará. Quando um thread termina ele chama o: thread-exit.

Se um thread quiser esperar até que outro thread termine o que está fazendo, ele chama thread-join. Se um thread quiser abrir mão da CPU para outro usar ele chama thread-yield.

Para proteger dados em relações a threads, por causa de processos pai e filho, devemos:

+ Threads POSIX: Padrões de chamadas de threads → Pthreads

Tabela: Pthread-create

	<u>↳</u> retorna o identificador do novo thread
" -exit	
" -join	<u>↳</u> identificador do thread a esperar é um variável
" -yield	
" -attr-init	cria e inicializa uma estrutura de atributos do thread.
" -attr-destroy	Remove o thread que os thread criaram, eles ainda existem

+ Implementando threads no espaço de usuário

↳ 1^a vantagem: o pacote de threads no nível de usuário pode ser implementado em um SO que não dá suporte aos threads.

↳ Quando ele é criado no espaço de usuário, cada thread precisa ter sua própria tabela de threads: armazena o estado dos threads, seu contador de programa, ponteiro de pilha, etc. Tudo que precisa para voltar ao mesmo lugar depois.

↳ 2^a vantagem: realizar o切换amento de threads com algumas instruções é mais rápido que deixar o controle para o núcleo (modo Kernel). Coisas contra:

↳ 1^a como os chamados bloqueantes não implementados, não dá pra deixar que uma thread faça uma chamada de sistema (porque vai parar todos os outros threads).

↳ solução: se for fazer um read(), o select() verifica a disponibilidade dos dados, evitando system call.

~~Torna better
fazer system calls~~ Sem no UNIX uma chamada **SELECT** que retorna com de novo

Seg Ter Qua Qui Sex Sáb Dom

Chamada será bloqueada, se não houver select estiver presente ela faz a chamada corrente quando da sua vez, ou seja, não fará bloqueado (esperando E/S). Se não for, outro thread é executado. O código colocado em torno da chamada de sistema para fazer isso é o socket ou wrapper.

2º Os threads têm que voluntariamente obter mão da CPU, pag dentro de um único processo não há interrupções de relógio.

3º (pior) programadores desejam threads nas aplicações, em vez mais fazem ^{demanda de sistema} threads no sistema.

+ Implementando threads no núcleo: Kernel

não há necessidade de um sistema de tempo de execução como no do usuário, não há uma tabela de threads para cada processo, sim uma tabela com todos. - Quando o thread que criou ou destrói uma thread existente, ela faz uma chamada de núcleo que faz a criação ou destruição atulizando a tabela.

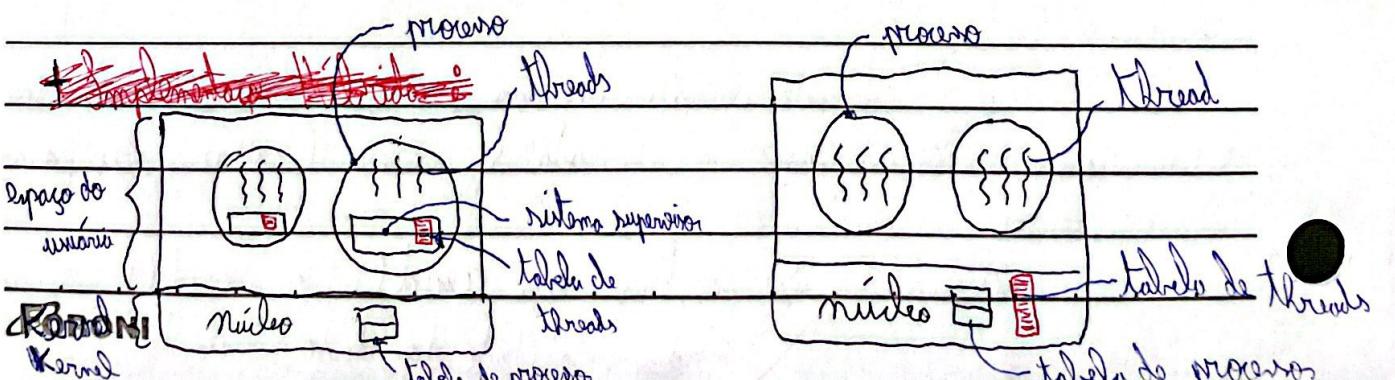
Todas as chamadas que poderiam bloquear uma thread são implementados como chamadas de sistema, quando uma é bloqueada o núcleo pode executar outra thread do mesmo ou de outro processo.

↳ Como o custo de criar e apagar threads no núcleo é maior elas se reutilizam, ou seja, quando sai apaga só destino, quando povoar, só ativa de novo.

↳ Threads de núcleo não precisam de qualquer chamadas de sistema nem são bloqueantes, além disso se uma thread provoca uma falta de página (o arquivo não está na RAM) o núcleo pode executar outras threads enquanto processa as exceções).

↳ Desvantagem: o custo das chamadas de sistema é alto, assim se elas forem frequentes há uma sobreexa.

↳ Mas ainda há os problemas (pausas, filhos, ...)



+ Implementação Híbrida: Uma abordagem que dá o máximo de flexibilidade é usar os threads de núcleo e entre multiplexar os threads de usuário em alguns ou todos eles. Assim o núcleo está só com os threads de usuário e os exclui. Um thread de usuário pode ter múltiplos threads de usuário multiplexados, os quais são criados, destruídos e escalonados como threads de usuário em um SO sem capacidade para múltiplos threads. Cada thread de usuário tem um conjunto de threads de usuário que se reservam para usá-lo.

+ Atividade pelo Escalonador: ideia Dóris: o núcleo notifica que aconteceu (thread bloqueado para esperar por algo) como com um signal, isso chama-se upcall. Então o sistema de tempo de execução pode reescalonar os seus threads, marcando o atual como bloqueado e pegando outro da lista de prontos, em processadores virtuais. (O que não segue o princípio fundamental dos carros: m opções sendo e m+1 pode chamá-las, mas m não deveria poder chamar versões de m+1 (o que ocorre nesse caso)).

+ Threads pop-up: ao invés de ter um processo bloqueado esperando algo, é quando algo acontece que o thread é criado. Como todos são criados igualmente (sem precisar retornar registradores e pilhas) eles podem ser criados rapidamente → diminui a latência entre a chegada da mensagem e o começo do processamento.

Os threads pop-up no núcleo é mais fácil, pois pode facilmente aceder todos os tabelas e dispositivos de E/S, mas se tiver um erro pode causar mais danos.

+ Convertendo código de um thread em multithread: é difícil, pode dar problemas com os variáveis globais (reservadas por outros threads).

Solução: variáveis globais privadas! Forma de exibir na pg 24 → item 2

2 - muitas bibliotecas não são reentrantes, ou seja, não foram programadas para ter uma segunda ^{execução} comando de rotina enquanto a outra não foi concluída.

3 - solução: ou você reverte toda a biblioteca, ou acrescenta 1 list que mostra se a biblioteca está sendo usada, evitando sobreescrita de dados.

4 - endereçamento de variável.

/ /

Processo: (VÍDEO AULA)

Seg	Ter	Qua	Qui	Sex	Sáb	Dom
-----	-----	-----	-----	-----	-----	-----

X Espaço de endereçamento

1) Texto: código executável (Text)

2) Dados: as variáveis (Data)

3) Pilha de execução. (Stack)

↳ controla a execução do processo

↳ empilhando chamadas de procedimento, etc

Stack

FFFF

GAP

Data

Text

0000 → relativos ao endereço

do processo (o 0000 mesmo é do SO)

→ processo CPU-bound: tempo de exec é definido pelo processador

→ processo I/O-bound: " " " " " pela duração das operações de E/S

X Escalonador (Processo que escala o processo que não usa a CPU)

↳ Disparante (dispatcher): Ativado, armazena e recupera as info dos processos

↳ Escalonador (scheduler): Escala a tarefa que usa o processador

↳ É chamado quando: um processo é criado, depois de fim ou é finalizado

X Threads

↳ de usuário (endomento):

Possível: A₁ A₂ A₃ A₁ A₂ A₃

↳ Mas, da pra fazer o mapeamento entre threads do núcleo

↳ Possível: A₁ B₁ A₂ B₂ A₃ B₃ (o SO mapeia os threads só o processo)

Possível: A₁ B₁ A₁ A₂ B₁ B₂

↳ de núcleo

Possível: A₁ A₂ A₃ A₁ A₂ A₃

↳ híbridas

mapeamento de m (Threads do núcleo) para

m (Threads do usuário)



2 JOBS = 20 minutos de CPU + 50% do tempo

↳ 1 JOB 20 min de CPU + 50% de E/S = 40 min

↳ 2 JOB

=

= 40 min

80 min → sequencial

Em paralelo:

A [20 min] [20 min] ;

B [20 min]

FORONI

Seg Ter Qua Qui Sex Sáb Dom

Questionário Semana 2.

$$\left. \begin{array}{l} \text{frequência} \\ \text{Uso da CPU} = 1 - p^m = 1 - 0,5^5 \\ = 0,968 \therefore 3,2\% \text{ liso} \\ \hookrightarrow = 1/32x \end{array} \right\}$$

1) Multiprogramado - grau 6

40% de espera

$$\text{Utilização} = 1 - p^m$$

$$U = 1 - 0,4^6 = 1 - 0,004$$

$$3 \cdot 2^6 = 192$$

$$U = 99,59\%$$

2) RAM = 4000 MB

$$SO = 512 MB \quad \text{PROCESSOS} \quad 256 MB$$

$$3488 \quad 1256$$

$$13,625 : 13$$

$$U = 1 - p$$

$$0,99 = 1 - p^{13}$$

$$p^{13} = 0,01$$

$$p = \sqrt[13]{0,01}$$

$$p = 0,70 \therefore \sim 72\% \text{ "a"}$$

3 e 4 alternativas

\hookrightarrow Em geral o manipulador de interrupções é montado em assembly porque o linguagem de alto nível em geral não permite acesso ao hardware da CPU.

\Rightarrow Processos podem passar de bloquedo para em execução se quando chegar o E/S a CPU estiver livre.

5) 2 JOBS \rightarrow 20 minutos de CPU cada

\hookrightarrow se 50% do TEMPO é em espera de E/S

cada JOB dura 40 minutos (modo CPU, modo E/S)

\therefore 80 minutos para o 2º terminar (repeticionalmente)

$$\text{Utilização da CPU} = 1 - 0,5^2 = 0,75 \Rightarrow 75\%$$

0,75 - 40 minutos (prog. dando a CPU)

(@)

$$1 - x$$

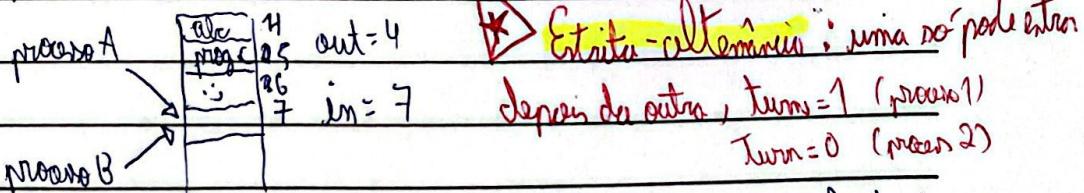
$$x = \frac{40}{0,75} = 53,3 \text{ minutos!}$$

FORONI

Comunicação entre Processos: IPC (Interc Process Communication)

→ também se aplica a threads de diferentes processos ou no escalonamento.

- Condições de Corrida: dois processos leem o mesmo endereço de memória como livre e tentam escrever nele.



↳ Região Crítica: Quando um processo quer acesso a memória compartilhada ele entra em região crítica, assim, deve-se impedir outros processos de acessá-la (Bloqueando-a).

1. 2 processos podem estar simultaneamente dentro de sua região crítica.
2. Nenhuma suspensão pode refletir o resultado da execução da CPU's.
3. Nenhum processo executando fora de sua região crítica pode bloquear outro querendo entrar.
4. " " deve ser obrigado a esperar eternalmente para entrar em sua região crítica.

Exclusão Mútua: um processo executa um de cada vez a região crítica compartilhada.

* **Exclusão Mútua com espera condicional:** enquanto um processo estiver ocupado trabalhando na região crítica, nenhum outro entrará nela causando problemas. (propostas para isso?)

① **Disabilitando Interrupções:** O processo desabilita interrupções logo após entrar em sua região crítica → ponto ativo: se ele nunca mais habilitar? se tiver mais de 1 núcleo?

(softwares) ② **Parâmetros do tipo Trava (Lock):** Se uma variável lock = 0 (não tem ninguém em região crítica), lock = 1 (tem) → não resolve, está sujeito a condição de corrida! O li trava / Alguém trava e quem

espera → ③ **Solução de Patterson:** uso funções enter-region () e leave-region() executa/Busca

④ **Instruções TSL (Set and Test Lock):** mult-núcleos, TSL RX, LOCK → premia quem tirou
Isso faz hardware para impedir acesso ao barramento de memória por outras CPU's, enquanto o ativer em sua região crítica. Intel x86 usa XC HG, é essencialmente o mesmo que TSL.
espera condicional: quando um processo quer entrar em sua região crítica, e é impossível, ele esperará em um lugar até que o processo desabilite.

* **Dormir e Acordar (Sleep e Wakeup)**

Problema do produtor-consumidor com buffer limitado:

produtor põe coisa no buffer e consumidor tira. Se o buffer estiver cheio o produtor dorme até que o consumidor tire. Se o consumidor tira 1 ou mais, se o buffer estiver vazio, o consumidor dorme até que o produtor coloque algo.

FORONI

Pode ocorrer condições de corrida se haver uma variação em que grande quantia, saia lá no buffer. Se o consumidor lê o m , foi parado pelo encodador, o produtor põe 1 em m e todo mundo vai pegar para o consumidor, como ele não estava dormindo o sinal se perde, aí voltando ao consumidor que vê que $m = 0$ ele dorme. assim ambos dormiram.

↳ solução: acrescentar um bit de espera pelo sinal acordos \rightarrow quando, por exemplo, o consumidor vê o sinal que é 0 e tenta adquirir, ele verá que é 1 o bit e ficará acordado, ler o novo e limpar o bit. \rightarrow problema: seria necessário mais bits para maior processo!

\rightarrow gerar: contadores
 \rightarrow fijar: mutex

\rightarrow recurso liberado \rightarrow que é um recurso

\rightarrow não há recurso disponíveis

* Semáforos: pode ter o valor 0 se nenhuma condição de despertar for atendida ou 1 ou mais se houver alguma. semáforo ++ (up) \rightarrow semáforo -- (down) dormir

↳ Solução usando Semáforos: 3 semáforos: 1 full (nº de vagas livres), 1 empty (número de vagas ocupadas) e 1 mutex (evita o consumidor e produtor acordarem o buffer ao mesmo tempo), também usa TSL para verificar que uma CPU examina de cada vez o semáforo. \rightarrow aí usa semáforo binários para o problema do produtor-consumidor. down: verifica se o valor armazenado é > 0 , se for, gera um sinal de acordo. mutex: usado para evitar exclusão mútua (um processo de cada vez lendo/saindo no buffer). full/empty: evita o produtor de escrever em um buffer cheio e o consumidor nem sair \rightarrow implementado pelo linux

Futex = fast user space mutex; trouxe implementação no espaço de usuário, em que o núcleo só se envolve bloqueando processos se a thread estiver em uso.

\rightarrow e criando uma fila de processos que estão aguardando

+ PThreads e Mutexes: os mutexes estão implementados na pthread.

- pthread_mutex_lock \rightarrow tenta adquirir a trancagem e é bloqueado se a trancagem estiver sendo usada.
- pthread_mutex_trylock \rightarrow se falhar retorna um erro, aí a thread pode decidir o que fazer.
- pthread_mutex_unlock \rightarrow destrói a trancagem, dando espaço para 1 thread.

É o programador que tem que criar o mutex? \rightarrow pthread_mutex_init

+ Variáveis de Condicionais: usadas juntas com mutexes, permitindo que threads

sejam bloqueadas devido a uma condição específica.

Pthread_cond - (init, destroy, wait, signal, broadcast)

\rightarrow é bloqueado esperando sinal

\rightarrow sinaliza para outro thread \rightarrow o depõe

FORONI

Seg Ter Qua Qui Sex Sáb Dom

com
Sleep e
waterup

, por isso

ainda a

signal

funcionam

/ / faz a exclusão mútua automática, ~~que é o que não ocorre~~ o que não ocorre

* Monitor: uma coleção de rotinas, variáveis e estrutura de dados, de nível mais alto, reunidas em um tipo especial de pacote. Assim, os processos têm acesso às suas rotinas, mas não à estrutura interna.

↳ Um processo por vez pode usar o monitor, se o monitor tiver outro dentro, o processo que o chamou é bloqueado.

↳ Para um processo que está dentro do monitor ser bloqueado, ele deve usar as variáveis de condição.

Em forma é diferente: usa as rotinas wait e notify (não são objetos, a condição da corrida).

Problema: não é em C, problema de monitor e semântica: em um sistema distribuído (cada CPU com própria memória...) → semântica não de muito bom nível e monitores não são utilizáveis.

* Fila de Mensagens: (Computadores diferentes)

↳ usa send (destino, & message) e receive (source, & message)

→ pode bloquear o processo de receber algo, ou retornar um erro.

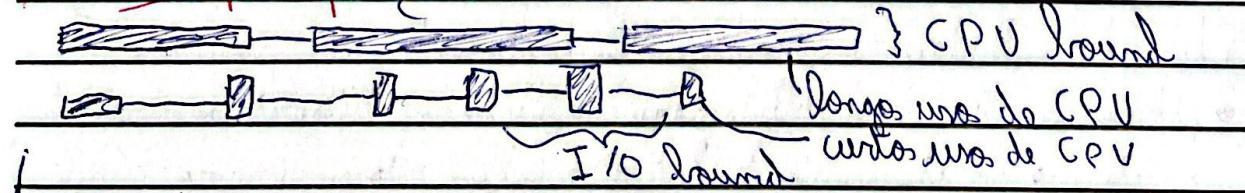
É implementado em rede de computadores, ai o PC pode mandar um receive para informar que recebeu a mensagem. Sem caixa postal e rendezvous (encontro marcado) → deixa o send e receive sincronizados quando bloqueio.

* Barréria: Espera todos os processos acordarem para que possam prosseguir

microondas

~~caso de máquina que fizer~~

Escalonamento



↳ Algoritmo não preemptivo: envolve um processo para executar e o deixa executando até ele ser bloqueado.

↳ Algoritmo preemptivo: envolve um processo e o deixa executando até um certo tempo (é necessário ter um relógio)

FORONI

por hora

Seg	Ter	Qua	Qui	Sex	Sáb	Dom
-----	-----	-----	-----	-----	-----	-----

/ /

nova

T de retorno

uso da CPU

Tempo de resposta
proporcionalidade

comprimento
dos processos
prioritidade

Categoria de Algoritmos de Escalonamento

- ① **Lote**: (ex banco) não há pessoas esperando imediatamente, mas não preemptiva ou com longo período → melhora o desempenho
- ② **Iterativo**: (ex servidores) algoritmos preemptivos são usados para evitar que um processo tome conta da CPU.
- ③ **Tempo real**: tipo servidor, como não é de uso geral, normalmente a aplicação não precisa de preemptivo → bloqueio instantâneo.

↳ **Leto Janela**: janela entre processos (novo da CPU), aplicação de política e equilíbrio "não preemptivo"

* Escalonamento de Sistemas em Lote: → first - come, first served

→ "Janela mais curta primeiro" (tempo considerado é chegar ao mesmo tempo)

→ "Tempo restante mais curto primeiro" (versão preemptiva dessa)

* Escalonamento de Sistemas Iterativos (um intervalo: Quantum)

↳ **Contexto** = mudar de processo que está executando

em termos de 20ms e 50ms é o ideal

{ Quantum muito curto: muitos contextos e reduz eficiência da CPU } circular

{ Quantum muito longo: resposta ruim e solicitações curtos }

→ Escalonamento por prioridade: cada um tem sua prioridade, se um com alta prioridade estiver dormindo, a prioridade dele pode diminuir a cada escalonamento, de fórmula interessante: $1/h \rightarrow$ fomos do último quantum que o processo usou onto de 1 ms de 50 ms = prioridade 50, se usou 25 ms = prioridade 2, ...

pro cada nova
chocamento da
prioridade pro
processo
2, 4, 8, 16, ...
n incremento

↳ dá pra agrupar por classe e fazer círculos dentro delas.

→ Multitarefa, → Processo mais curto em seguida (com aging) → calculo usando média ponderada

→ Escalonamento Garantido (promete e cumpre, tipo promete uso da CPU)

→ Escalonamento por Loteria: dá bilhete ao processo e sorteia os que ganham receberem o tempo (do quantum) da CPU, processos com + prioridade recebem mais bilhete, aumentando sua proporção de uso da CPU ao longo do tempo $un1 = A, B, C, \dots$
 $un2 = E$

→ Escalonamento por fração justa: leva em consideração mais de 1 usuário processos A, E, B, E

* Escalonamento em Intervos de tempo real: M evento, período de um evento $i = P_i$

segundo de $i = C_i$ → periódico

se pode escrever de: $\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$, se for verdade ele é escalonável.

FORONI

de escalonamento

- **Políticas versus Memória:** separam a política de escalonamento, deixando isso a cargo do processo pai, por exemplo, que establece o escalonamento de seus filhos através de prioridades dos escalonadores. Assim a política de escalonamento fica com o processo pai, mas é o escalonador que está no núcleo que faz o escalonamento (memória).
- **Escalonamento de Threads**

Se for threads de usuário: o núcleo não sabe, então ele faz como sempre faz.
 Se for threads de núcleo: o núcleo sabe das threads, assim pode alterar entre alternância de threads de diferentes processos [não só foi andando no vídeo aula].

(O escalonador de thread dentro do processo que executa).

Setmana 5

- Problemas com a Comunicação de Processos

O → Janela dos filórios: De fato: pessoa → pegar hambú direito → pegar hambú esquerdo →
 | → come → pessoa
 | → pode ocorrer deadlock (empasse) ou starvation.

De fato com matriz: só um come de cada vez

O Leitor e Escritor: Quando escritores entram em a base de dados, deve ser bloqueada

↳ Leitores: se tiverem de bloquedas: deve ser bloqueada para escritor escritor
 | → se não | → se tiver um leitor lá, pode entrar (ou não de um)
 | → escritor se tem leitor dentro, se não de bloqueio a base

Início: o escritor pode entrar, não entrando se for bloquendo, e continua entrando leitores

FORONI ↳ Resolução: se um escritor chegar, impede leitores de entrar até que ele sair da BD

Deadlocks (impare)

Correm em recursos compartilhados; → os processos ficam parados sem continuar, tanto em nível de hardware e software.

↳ Deadlocks ocorrem quando processos obtêm acesso exclusivo aos recursos.

→ Recursos:

- Preemptivos: podem ser retirados sem prejuízo. Ex: memória e CPU
- Não Preemptivos: não podem ser retirados do processo, causam prejuízo. Ex: impressora

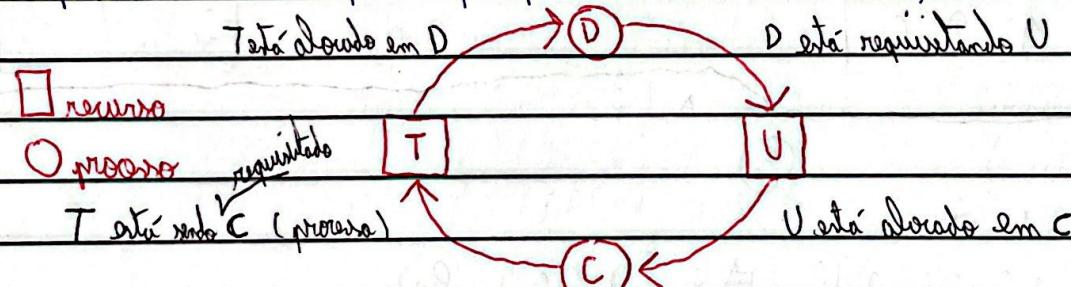
1. Requisição do Recurso
2. Utilização do Recurso
3. Liberação do Recurso

{ Deadlocks ocorrem normalmente em não preemptivos.
↳ pq de se larga o recurso até acabar de usar

Condições para Deadlocks

Processos estão em deadlock se cada processo estiver esperando um evento que somente o outro processo pode causar.

- Exclusão Mútua: um recurso só pode estar aloocado para um processo em determinado momento
- Hold e espera (Hold and wait): processo que já tem recursos querendo mais
- Não preemptivo: recursos alocados não podem ser retirados, só o próprio processo os libera.
- Espera Circular: um processo espera por recursos alocados em outro, em uma cadeia circular



O deadlock circular ocorre porque um processo já possui um recurso e quer outro (hold and wait), só podendo continuar seu execução quando ele o obtiver (não preemptivo).

Estratégias Para Deadlocks

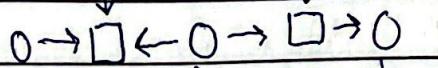
- Ignorar o problema: é bom se a frequência com que ocorre for baixa comparada a outras falhas ou se é muito caro resolver o problema. (algoritmo do Gostinho)
- Detectar e Recuperar

• Detectar e Recuperar (permite que deadlocks ocorram, tanto detectar e resolver a situaçao.)

• Detecção com um recurso de cada tipo.



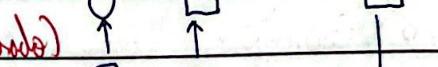
• " " vários recursos de cada tipo.



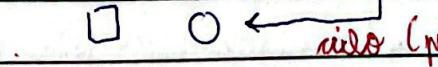
• Recuperação por meio de préemprego



• " " " rollback (volta ao ponto)



• " " " " dimensão do processo.



círculo (probabilidade de ocorrer deadlock)

Detectão de Vários Recursos

Sóis processos

$P_1 \rightarrow$ uma impressora

$P_2 \rightarrow$ uma 2 unids de fita e CD-ROM

$P_3 \rightarrow$ uma um plotter e 2 impressoras

Recurso Existente

$$E = (4, 2, 3, 1) \quad \begin{matrix} UF \\ P \\ I \\ CD \end{matrix}$$

Recurso Disponível (disponivel)

$$A = (2, 1, 0, 0) \quad \begin{matrix} UF \\ P \\ I \\ CD \end{matrix}$$

Matriz de Alocação

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \quad \begin{matrix} UF \\ P \\ I \\ CD \end{matrix}$$

Matriz de Requisição
Recurso Disponível

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{matrix} UF \\ P \\ I \\ CD \end{matrix}$$

$$A = (2, 1, 0, 0) \quad \begin{matrix} UF \\ P \\ I \\ CD \end{matrix}$$

$$\begin{matrix} UF \\ P \\ I \\ CD \end{matrix} \quad \begin{matrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{matrix} \quad \begin{matrix} P_1 \\ P_2 \\ P_3 \end{matrix}$$

recursos

$$A = (2, 1, 0, 0)$$

$$\textcircled{1} \quad A = (2, 1, 0, 0)$$

$$\textcircled{2} \quad A = (2, 2, 0)$$

$$A = (2, 2, 2, 0) +$$

$$\textcircled{3} \quad A = (4, 2, 2, 1)$$

$$M_{\text{d. requisições}} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{matrix} P_1 \\ P_2 \\ P_3 \end{matrix} \quad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{matrix} P_1 \\ P_2 \\ P_3 \end{matrix}$$

Lerfilho Paráiso :- Início de antecipação os recursos o serem utilizados

- Quando processos por deadlocks → ativação do modo CPU, ou seja, se o modo CPU estiverativo com parado ao nº de processos.

→ todos os recursos são liberados toda vez que uma requisição é feita → a cada K minuto

• Recuperação de Deadlocks → dps q detecta tem q corrigir?

↳ Por meio da preempção: retira o recurso de um processo (não preemptivo) e entrega-lá a outro → depende muito de qual processo que é [difícil / impossível]

↳ Rollback: → o endereço de cada recurso é armazenado em um arquivo de verificação (clockpoint file) → o tutto feito dps dele é perdido
 ↳ Quando ocorre deadlock, o processo volta ao clockpoint anterior sem o recurso que causou o deadlock; o qual será dado a outro recurso.

↳ Prolongação do Processo: um ou mais processos em deadlock são interrompidos → melhor solução para processos que não causam muito prejuízo
 Ex: compilador (sem problemas); atualização de Base de Dados (problemas)

* algoritmo do banqueiro é pouco usado na real pq é difícil saber q recursos não usar
 ↳ Evitar Dinamicamente: alocação de recursos à medida que se fazem reservas; esforço maior → alto custo overhead

(Algoritmo [notação de Estados Seguros e Insseguros])

• Banqueiro para um único tipo de recurso.
 • " " " vários tipos de recurso. → deadlock possibilidade de ocorrência de deadlock
 [é o layout das matrizes?]

Estado Seguro: Se houver uma ordem no qual todo processo irá terminar

Estado Insseguro: Não garante que todos os processos terminarão corretamente

	Han	Marx								
A	3	9	3	9	3	9	3	9	3	9
B	2	4	4	4	0	-	0	-	0	-
C	2	7	2	7	7	7	0	-	0	-
free:	3		1		5		0		7	

Exemplo
MESMO
TIPO de
RECURSO

Algoritmo do Banqueiro: atende sempre o q. previsor de menor recursos 1º, quando é

" " " com prioridade de um tipo de recurso
 available → A = (1 0 1 0)

D	3 2 0 1	R =	0 0 1 1 0	→	D =	3 2 1 1 1	R =	0 1 0 1 0	⇒ A = (4, 2, 1, 1)
---	---------------	-----	-------------------	---	-----	-------------------	-----	-------------------	--------------------

C = recurso
alocado

R = recurso
remanescente

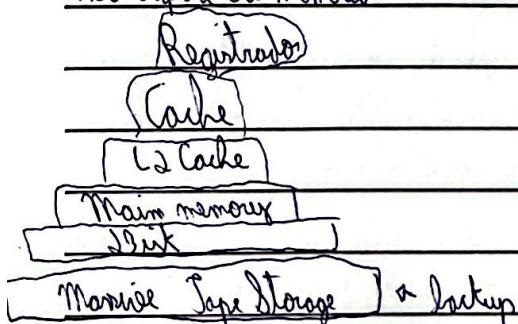
FORONI

Principais de Deadlocks (atascando suas causas)

- Exclusão mútua: Bloquear recursos usando spooling (não o printador tem acesso à impressora)
- Uso e espera: Processos que requerem todos os recursos para executar → fazer isso 1 cada vez (é difícil)
- Não preempção: Retirar recursos do processo é difícil sem causar prejuízo
- Espera Circular: Ordena numericamente os recursos; realizar solicitações em ordem numérica [permite usar só um processador]

Sexta-feira 6 → Gerenciamento de Memória

Hierarquia da Memória



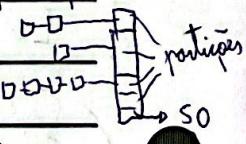
Tarefas do Gerenciador de Memória

- ↳ Gerenciar a hierarquia de memória
- ↳ Gerenciar espaço livre / ocupado
- ↳ Bloquear / localizar processos / dados na memória
- ↳ Controlar partes que estão em uso ou não
- ↳ Alugar: memória com processos se movimentam
- ↳ Silenciar: memória quando um processo termina

Swapping: Funcionamento entre a memória principal e a de disco & memória principal e código

Gerenciamento de Memória - Multiprogramação

- ↳ divide a memória em n partições de tamanho fixo
- ↳ Quando chega um sobraria ele na fila; o espaço que sobra não é utilizado
- ↳ endereçamento: como dar a cada ~~sistema~~ programa seu próprio espaço de endereço, de modo que o endereço 28 em um seja diferente do 28 em outro?



- * 2 registradores: base + limite
 - base → 20 → armazenado no endereço 20
 - limite → 60 →
- (circular em hardware)*

MMU → Memory Management Unit → converte o endereço lógico no endereço físico

Memória Particionada:

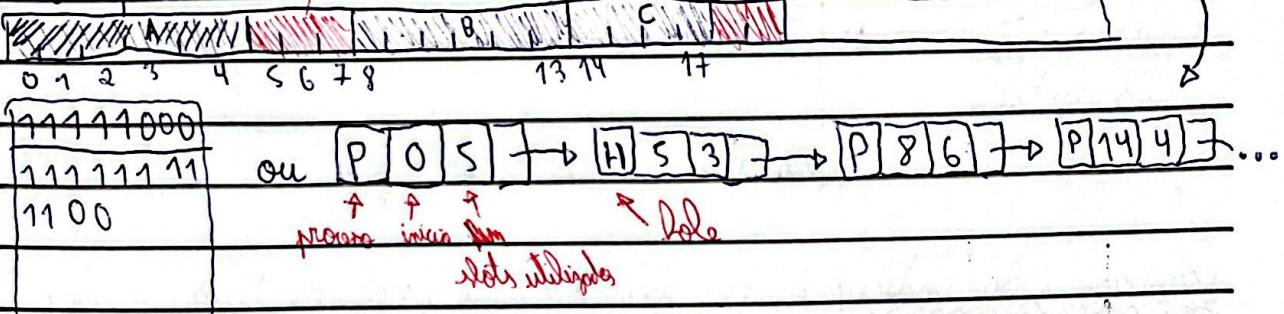
- Partição física (alocação estática): mais simples, tendem a desperdiçar memória
- " " . Partição dinâmica: complicado, para dar e tirar, otimiza a memória

FORONI

Swapping : → swap-out : da memória para o disco, em uma área de swap
 → swap-in : do disco para a memória

o Gerenciamento da Memória

Bitmap :



Algoritmo de Colisão :

Melhor encolha : encolher a área que mais se encaixa no processo (a menor possível)

Pior encolha : inverso → prova-se mais áreas para alocação de processo.

Próxima encolha : no próximo que couber.

Técnica de Memória Virtual :

→ faz a RAM parecer muito maior

é uma técnica que usa memória secundária (como um cache → p/p o tamanho de todos os processos juntos é menor que mem. principal) → só guarda o que realmente está usando na principal. Uso MMU para a conversão

Técnica :

1 + Paginagem : quebra o processo em páginas de 4KB

2 + Segmentação : segmenta a memória em blocos de mesmo tipo (dados, código, pilha)
 ↳ ai da pra ter tipos de acesso diferentes a cada um.

1 → página : unidade de transferência fixa no disco ; frames : unidades correspondentes na RAM

page fault : quando uma página que não está na RAM é referenciada

tabela de páginas : faz o mapeamento

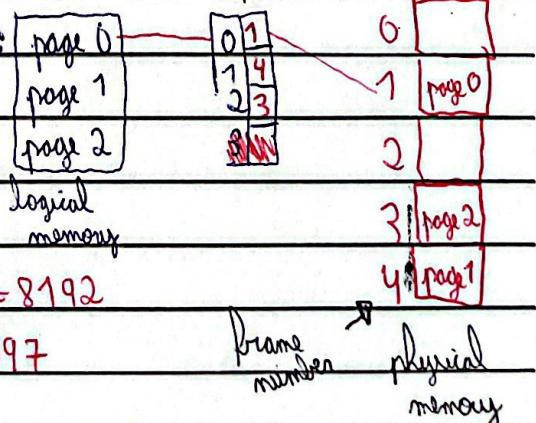
Ex: um sistema que gera 64K de endereços virtuais

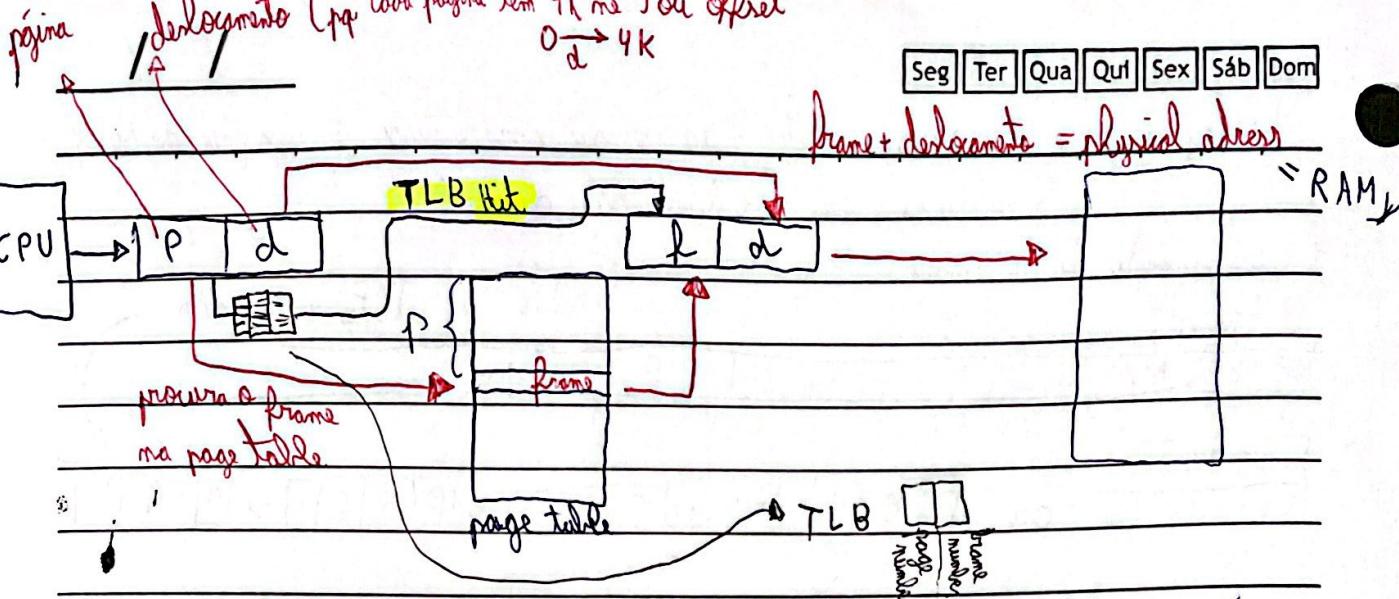
- MMU faz o mapeamento

- MOV REG, 5

- Suponha que esteja mapeado no 3º frame, remetendo em 8K = 8192

- Comendo endereço ao barramento é $8192 + 5 = 8197$





Componentes do Endereço: p: usado como índice para uma tabela de páginas
d: deslocamento de página ($p + d = \text{endereço físico enviado à memória}$)

Páginas maiores: leitura mais eficiente, tabela menor, mais fragmentação

"menores": "menos", "maior", "menos"

Componentes da Tabela: Page frame number; bit de residência (está ou não na RAM);
bit de proteção (0(R/w), 1(R), 2(exec)); bit de modificação (modificada ou não);
bit de referência (recentemente referenciado ou não); bit de cache (disponível ou não o address do page).

Onde Armazenar a Tabela: - array de registradores: se a memória for pequena (fica no hardware)

- No RAM: a MMU gera mapeia usando um dos registradores

→ Em uma memória cache na MMU: chamado de memória associativa (melhora o desempenho) hardware
(fica no hardware) cache da tabela das páginas mais usadas → TLB (Translation Lookaside Buffer)

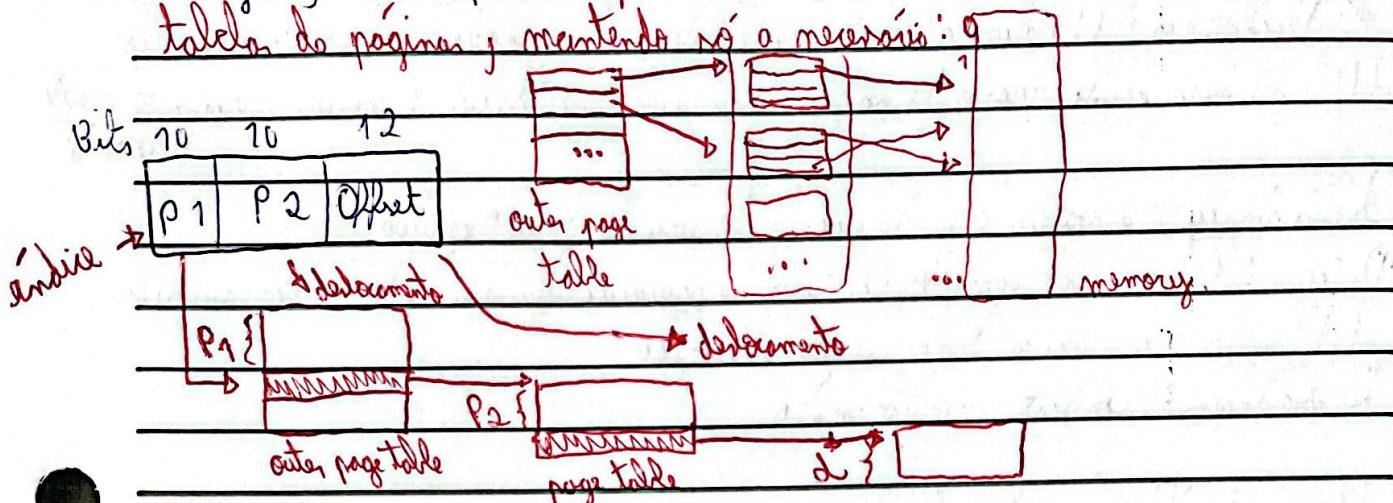
Fallas de Página: - Soft Miss: não está na TLB, mas tá na tabela de página

- Hard Miss: não está na TLB nem na tabela de página

Semana 7 → continuação paginação

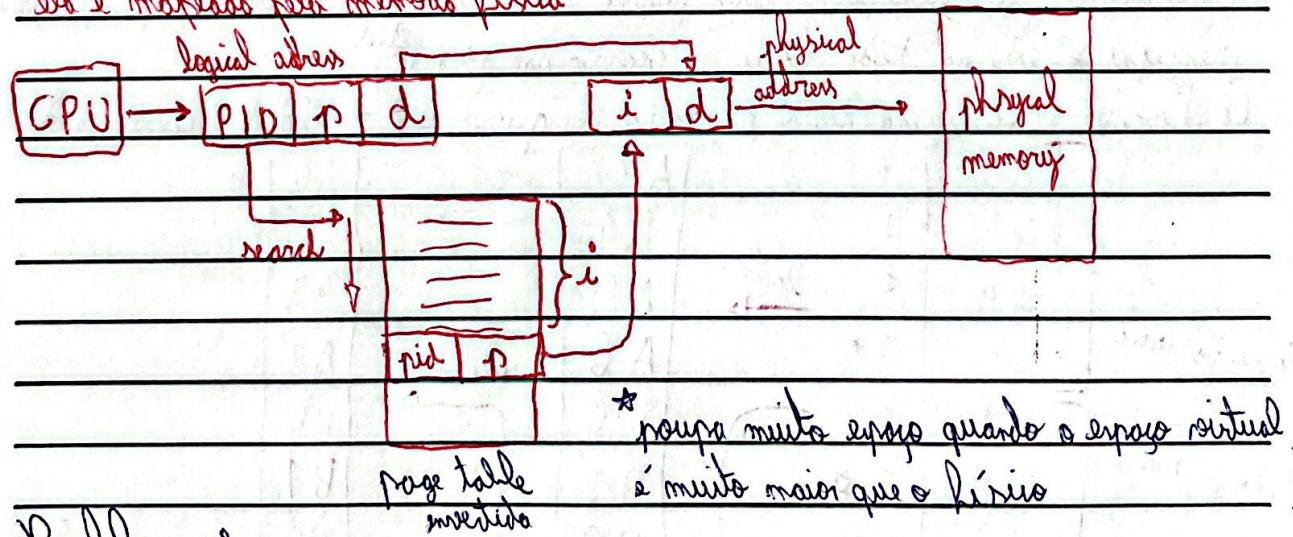
11

- Paginado Hierárquico: quebra o espaço de endereço lógico em múltiplas tabelas de páginas, mantendo só o necessário:



- Tabelo de Páginas em Flash (funciona bem com 32 bits, mas só com 64)
 - ↳ um número de página virtual é usado para função hash, que gera um endereço [# do pag virtual, # do frame, um ponteiro para o próximo]

- Tabelo de Páginas Invertida: ao invés de mapear pela memória virtual, ela é mapeada pela memória física



Problemas:

- Gasta muito tempo para ler a memória: solução?
 - usar a TLB para as mais usadas e se não tiver na page table invertida

Alocação de Páginas :

fixa → é simples

↳ cada processo tem um número máximo de páginas resis definido quando ele é criado

* Problemas : (1) número muito pequeno de páginas pode causar muita paginação

(11) número muito grande de páginas pode gerar desperdício (não que o processso precise de tanto)

Dinâmico : varia o nº de páginas durante sua execução

Vantagem : (1) processos com elevada taxa de paginação tem seu limite de pag ampliado

(11) e os com baixa taxa limite reduzido

Desvantagem : consumo monetário

Burco de Páginas : → quando vai rodar um processo

→ Pag. Simples = todas são carregadas para a memória principal

→ Pag. por Demanda = conexão com nenhuma pag na memória, assim que a CPU tenta executar gera page fault, o SO traz a pag que falta na memória e assim vai.

Política de Substituição → qual página desse tiro no momento em que a página principal já estiver toda cheia e ocorrer um page fault

lo Local : Considera apenas o processo em questão } Global : considera todos

A0	10	A0	A0
A1	7	A1	A1
A2	5	local → A2	A2
A3	3*	A6	A3
B0	9	B0	B0
B1	2*	B1	B1
B2	5	B2	B2

página do processo

pag do processo s

age

remover

ocorre falta

de página A6

* página mais antiga de A

página mais antiga que tem

FORONI (é a com menor número pag
que ocorre a falta)

Algoritmos de Substituição de Páginas

+ Algoritmo Ótimo: não existe, mas usado para fazer comparações

↳ cada pg é marcada com o número de intrusões que possui e sempre responde a com menor probabilidade de ser referenciada

+ NRU (Algorithm Not Recently Used Page Replacement)

↳ 2 bits pro SO mem = R (referenciado) + M (modificado)

00 - \tilde{n} R \tilde{e} M ; 01 - \tilde{m} R \tilde{e} M ; 10 - R \tilde{e} M ; 11 - R \tilde{e} M

R \rightarrow lido ou escrita, M \rightarrow escrita

R e M são atualizados a cada referência à memória, eles ficam armazenados na entrada do tableau de página

\rightarrow periodicamente o bit R é limpo

\rightarrow o bit M não é limpo pq o SO precisa saber se deve escrever a página no disco

+ FIFO \rightarrow simples mas pode ser ineficiente: tirar uma página que está sendo constantemente usada (\tilde{n} é muito utilizado)

+ Algoritmo da Segunda Chance (bit R + FIFO)

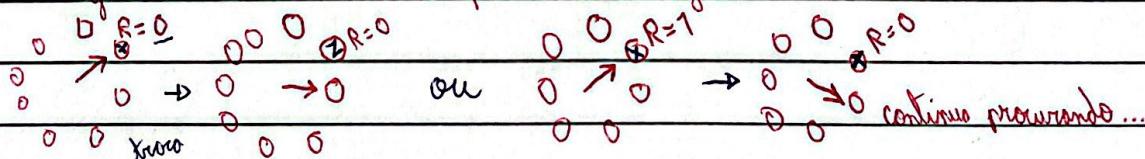
↳ inspeção do bit R da fila

\rightarrow se for 0 $\rightarrow \tilde{n}$ foi usada recentemente \rightarrow é trocada

\rightarrow se for 1 \rightarrow é colocada no final da fila com R=0 \rightarrow a busca continua [segundo chance]

+ Algoritmo do Relógio: melhoria do do 2a chance

↳ é igual o do 2a chance só que é um relógio



+ Algoritmo: Least Recently Used Page Replacement (LRU)

↳ páginas muito usadas recentemente provavelmente serão usadas no futuro; troca a que permaneceu em uso pelo mais tempo.

↳ alto custo = manter na memória uma lista encadeada com todas as páginas

que estão na memória com as mais recentes.

FORONI

- ↳ Implementação em Hardware: no nível do MMU (contadores em cada pg → o menor foi referenciado e saiu)
- ↳ Em Software: mesma coisa que no hardware
problema: como não contadores, eles não esquecem de nada, pg muito usadas no começo serão mantidas mesmo se não forem usadas depois.

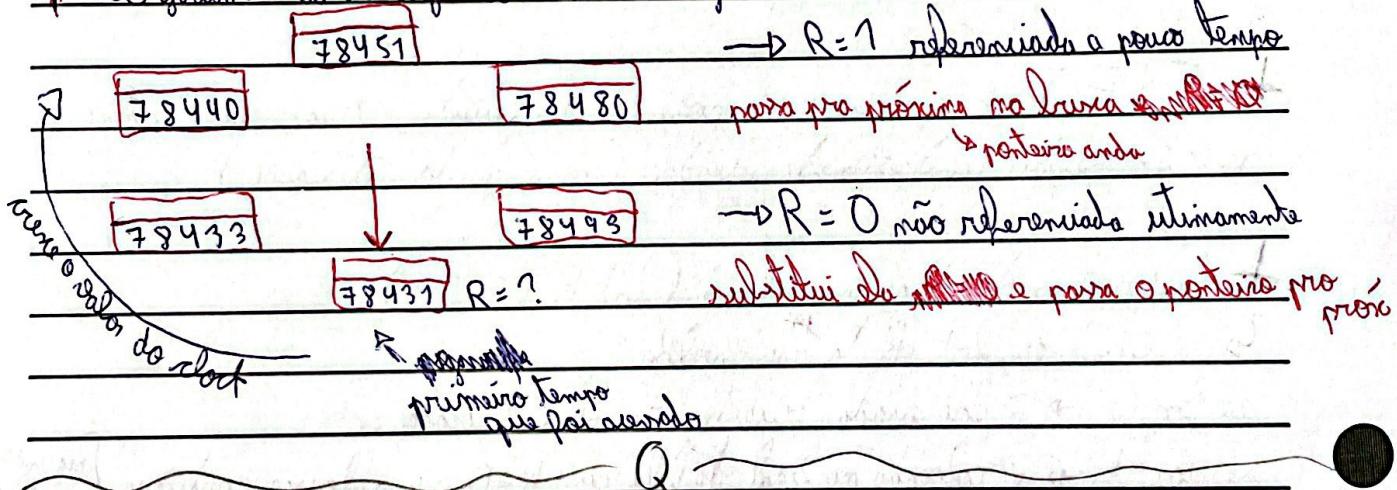
+ Algoritmo do Working Set \Rightarrow determina um conjunto de páginas que o processo utilizou em um determinado tempo T e as coloca na memória antes dele rodar.

↳ dá para estimar o número de páginas necessárias que o utilizou com base nas que ele usou quando foi interrompido (pré-paginagem)

- working set: o conj. de pag que o processo usou durante o último T segundos
- usa o bit R

antes do interrupt

+ Algoritmo Working Set Com Relógio (WSC)



Custo extra por página = 5 ns

TLB com 32 páginas [pág virtual, quadro pg física] = 1 ns

$$(1-x) \cdot 5 + X \cdot 1 = 2$$

$$5 - 5x + x = 2$$

$$-4x = -3 \Rightarrow x = \frac{3}{4} = 0,75\% \text{ da frequência do TLB!}$$

Aula Recurso → Gerenciamento de Memória

Registrador-base: endereço de início da página

“ - limite : contador com o tamanho da página

Se o PC é de 64 bits o endereço máx do memória virtual é de 2^{64}

• Como saber a página e o deslocamento? Ex:

- Se for de 4KB a página :

de deve um endereço lógico de 20.000:

$4000 = 111110100000$ → bits necessários para o deslocamento na página
o endereço: 12 bits

$20.000 = 100111000100000$

índice da página deslocamento $= (4, 3616)$

- Se for de 8KB a página

$8000 = 11111010000000$

O mesmo endereço virtual: 13 bits

$20.000 = 100111000100000$

índice da página 13 bits de deslocamento $= (2, 3616)$

xx

Fórmulas

$$\text{Uso da CPU} = 1 - p^m \quad [p: \text{fração de tempo que processo ficou operando por E/S}]$$

[m: número de processos]

→ fração de tempo desperdiçado pela CPU → se os programas ficassem esperando por E/S na metade do tempo, e tem 5 programas:

$$\text{Prob de 5 estarem esperando} = \left(\frac{1}{2}\right)^5 = \frac{1}{32} \rightarrow \text{a fração de tempo desperdiçado da CPU}$$

Instrução = 1 mseg ; falta de página = m mseg adicionais.

Tempo de duração das instruções de as faltas de página ocorrem a cada:

$$K_{\text{instruções}} = 1 + \frac{m}{x} \text{ mseg}$$

FORONI

Semana 10

Introdução → Sistema de Arquivos

- Vista do usuário: - interface
(alto nível)
 - como são nomeados e protegidos
 - operações que podem ser realizadas
- SO (baixo nível)
 - como arquivos são armazenados fisicamente
 - " " " referenciados (links)

• Nome de Arquivos

Linux é case sensitive, o Windows não
nome + extensão < 255 caracteres

• Estrutura de Arquivos

1 - Sequência não estruturada de bytes:

- para o SO não importa conjunta de bytes, não se importa com o conteúdo
- significado dado pelas aplicações

[Elevável] → usuário põe o que quiserem Ex: Unix e Windows

2 - Sequência de Registros de Tamanho Fixo (mais antigo):

Cada um com sua estrutura interna (80 a 132 caracteres)

Lectura e escrita realizadas em registros → cartão perfurado

3 - Cúspides de Registro (tamanho variado): mainframes antigas

cada arquivo possui um campo chave em uma posição fixa.

Consiste em uma árvore dado | → | dado | → |

Operação: lê o registro a partir de uma certa chave

SO decide onde colocar os registros não o usuário

Tipo de arquivo: arquivo regulares → info dos usuários

diretórios → para estruturar o sistema de arquivos

arquivos do sistema → /dev e /proc

→ ASCII: linhas de texto com CR-LF → mais portáveis e interoperáveis

→ Binário: todo arquivo não ASCII, estrutura interna conhecida apenas pelos

PROGRAMAS que os utilizam. Ex: programa executável, .c, .doc, ...

0

Atributos de arquivos: além do nome, todo arquivo tem outras info sobre ele, os atributos (ou metadados), varia de SO para SO. Exe: segurança (quem cessa), sendo criador dono, chmod, oculto ou não, arquivo de sistema ou não, tem lockup ou não.



Sistema de Arquivos e Directório

Criar os arquivos:

SO's antigo: acesso sequencial, leitura byte a byte (reg a registro)

SO's moderno: acesso direto, fuso de ordem por meio das chaves

→ métodos para leitura:

read: indica a posição do arquivo a ser lido, a partir dele a leitura é sequencial
 write → marca a posição corrente
 execute → interrompe write para depois na posição desejada, após é resumido

(Operações do Arquivo: (system calls))

→ Create: cria arquivo sem dados

→ Delete: " remove do disco

→ Open: SO lê os atributos do arquivo e luta os endereços de bytes na memória

→ close: libera o espaço ocupado por open e força com que o último bloco de dados seja escrito no disco

→ read: lê o arquivo para o buffer

→ write

→ append: escreve no final do arquivo

→ seek: para acesso direto, procura dados.

→ get_attributes

→ set_attributes

→ rename

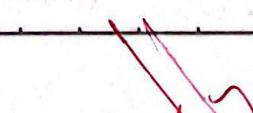
Arquivos Mapeados na Memória

Alguns SO's mapeiam arquivos no espaço de endereçamento (virtual) do processo.

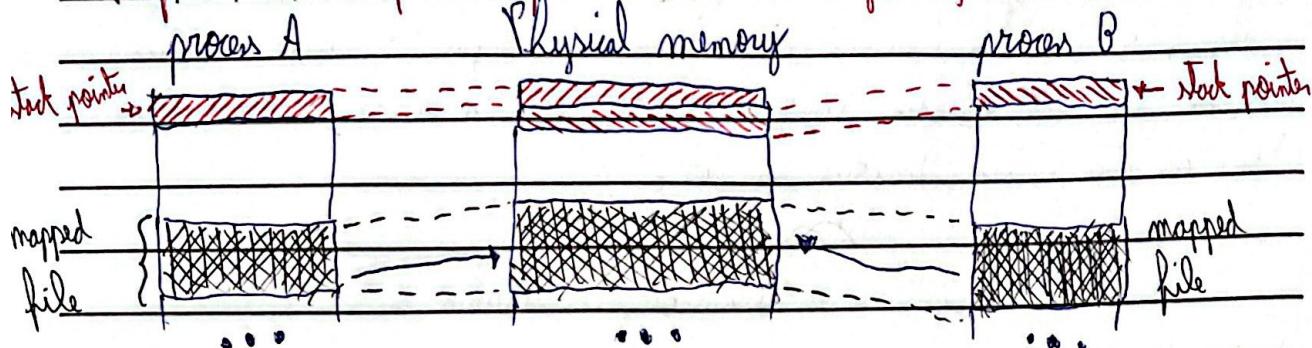
→ mais rápido, lido/exito como arquivo de bytes na memória

→ arquivos modificados gravados no disco ao final do uso.

→ System Calls: map e unmap.



Mapamento de Arquivos → função melhor com segmentação



O arquivo é compartilhado por mais de um processo, podendo ser editado.

Ex: google docs compartilhado, bibliotecas, ...

- vantagens: compartilhamento → trabalho colaborativo

- problema: uso do espaço virtual para o arquivo

- microscopia

- arquivo maior que o espaço virtual → alocação por partes

xxx

• Diretórios = não arquivos que mantêm a estrutura do sistema de arquivos

↳ Podem ser organizados = em nível único, em níveis, hierárquico (main comum)

↳ Caminhos = absoluto / relativo

↳ Links: → soft link: ponteiro para outro arquivo ou dir, se remove o arquivo que só tem o link

→ hard link: para cada link exerce um contador, só dá pra apagar o arquivo se o contador for 0.

* Unlink (para hard link) → remove o link e decrementa o contador

dos for 0.

Domingo 11 → Implementação do Sis de arquivos

* Há pra ter um dual Boot → 2 partições

- Setor 0 do disco é o MBR = Master boot record (modo pra iniciar o computador)

partições → [MBR] → [partição 1] → [partição 2] → [partição 3] → [partição 4]

MBR	Superblock	Free Space Map	I-nodes	Root dir	Files and directories
partição	partição 1	informações da	estrutura de	endereço do	demais arquivos
de	partição 2	partição 2	dados para cada	diretório	e diretórios
partição	partição 3	partição 3	arquivo	raiz	
	partição 4	partição 4	(um pra cada)	(tipo de arr.)	
	ID do Sistema de arquivos	Bitmap ou			
		Lista de			
		partes			
				do int. de	
				arquivos	

FORONI

Problema: Considere a remoção de um arquivo no Unix.

Pontos:

- Remoção do arquivo de seu diretório → e se falta aqui? → os blocos não serão liberados
- Silseração do I-node para o conjunto de I-nodes livres → I-nodes com blocos não liberados → e não redimensionados
- Redimensionamento dos blocos livres no disco → aqui? → os blocos serão perdidos.

O Solução: journaling = técnica para criar robustez diante de falhas.

- ↳ mantém-se um: log - o journal (info sobre o que o sistema vai fazer, não de fato)
- ↳ se o sistema falhar antes de executar o trabalho, faz depois da inicialização.
- Bloco no NTFS (Windows) e ext 3 (Linux)
 - ↳ ou seja: → grava o conj. de passos (logs) → desmonta tarefa → lê de novo o log → recupera o resultado.
 - ↳ quando a tarefa é concluída, apaga os logs.

* Desvantagem \Rightarrow devem poder ser readidas sempre que necessário sem causar danos.

ex: inclui novo bloco no final da lista de blocos livres. [é idempotente] \Rightarrow os blocos podem ser lidos.

ex: pergunta se os blocos já estão lá e, caso não, inclui-los [IDEMPOTENTE]

Alocação de Arquivos

1- Alocação Contínua

2- " com Lista Encadeada

3- " " " utilizando uma tabela de memória (FAT)

4- I-Nodes

este artigo no HD é
ainda é no CD-ROM

① Alocação Contínua: Criação de arquivo de forma contínua no disco

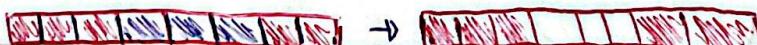
ex: Bloco de 1KB em um arq. de 50KB \rightarrow serão armazenados de forma contínua.

Desejável: é simples

↳ um vez que o primeiro bloco é referente, logo é só continuar lendo, rápido na leitura

Desvantagens: com o tempo o disco se torna fragmentado; recuperação após falha difícil

rq: tem que conhecer o tamanho exatosamente do arquivo para prender o "lugar".

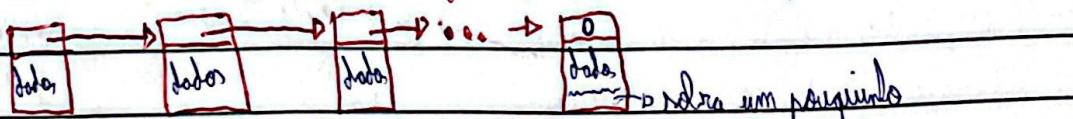


↳ Isto resulta? "lugar".

FORONI

② Alocação com lista encadeada

1ª palavra é o endereço do bloco seguinte; no fim do arquivo o endereço é 0



Vantagem = Não se perde espaço com fragmentação (só no último bloco)

Desvantagem = Custo que o endereço do diretório armazena endereços de 1º bloco.

Desvantagem = alocar os arquivos é desperdício, alocar diretório é lento e tem que alocar os endereços em cada bloco.

③ (Aloc c/ lista encadeada onde estático na memória (File Allocation Table) FAT

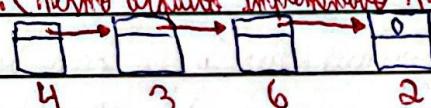
↳ Os ponteiros são colocados em uma tabela na memória principal

↳ Toda coluna de endereços fixa na memória, alocar diretório mais rápido

clínica

↳ diretório armazena endereços de bloco principal

Ex.: (número arquivo implementado no ① e no ③)



0		
1		
2	-1	+ Termina aqui
3	6	
4	3	+ começa aqui
5	2	
6	2	

Desvantagem = Toda a tabela deve estar na memória; um disco de 200 GB

com blocos de 1 KB → tabela com 200 milhões de entradas, cada 1/4 byte → 500 MB de RAM. :)

④ I-Nodes = cada arquivo tem sua estrutura

fatidico

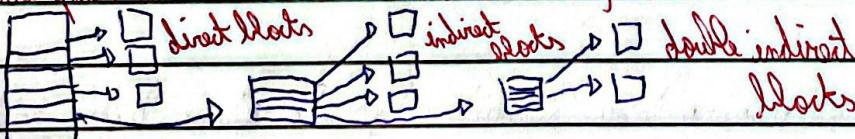
↳ se tem o I-Node de um arquivo, dá pra encontrar todos os blocos desse arquivo

Desvantagem = O I-Node é carregado na memória quando o arquivo é aberto; o espaço na memória ocupado é proporcional ao número de arquivos q podem ser abertos no mesmo tempo. ≠ FAT

↳ cada I-Node ocupa n bytes e no max. K podem ser abertos - total ocupado é K.n bytes.

Desvantagem = cada I-node tem espaço para um número fixo de endereços de disco, o tamanho do arquivo pode ultrapassar esse limite → solução → usar o último para blocos c/mais livre

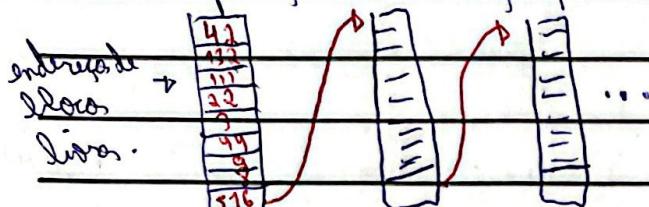
ig



Grenamento de Blocos Singulares

- ① Lista ligada de blocos livres = primeiro endereço de bloco livre é armazenado em um local específico do disco ou em cache (memória).
Problema = ponto fixo; caso precise carregar o bloco

- ② Lista ligada de blocos : é um bloco de endereços de blocos livres, o último endereço é o endereço para o próximo bloco.



→ Somente um é correto na memória = quando um arquivo é criado e escrita os blocos livres o novo bloco é lido do disco para a memória
 = quando um arquivo é apagado e os blocos não liberados; adiciona ao bloco de ponteiro na memória; quando ele estiver completo é escrito no disco.

Vantagem = conserva um bloco livre na memória, faz funcionamento muito rápido entre o disco e a memória. É requer mais espaço de disco que o bloco.

Pontagem = requer menos espaço de disco que o bloco.

- 3) Mapa de blocos = depende do tamanho do disco, disco maior mapa maior
 Um disco com n blocos requer um mapa com m blocos → fica na memória principal

1 - Bloco livre, 0 - Bloco usado

Vantagem = facilita alocação contígua

Desvantagens = torna-se lento com disco que o disco

(Questionário Semana 11 °)

Semana 12 → Introdução a I/O = E/S

SO controla: - comando para emitir instruções (read, write, etc).

- interruptor interrupções (ex: leitura do disco fez).

- tratar erros

API

- Usa uma interface única para todos os dispositivos -

- Elemento base para o SO (ex: drivers, gerenciador de tempo, ...)

Típos de E/S, podem ser divididos em:

- 1 → Típo de Conexão
- 2 → " " Transferência de dados
- 3 → " " Compartilhamento de conexão

1) Módulo de E/S x Periférico

Serial X Paralela

→ Típica linha de conexão 

Vantagens = mais barata, mais lenta que o paralelo, relativamente confiável, usado em dispositivos mais leves e baratos (impressoras e terminais)

→ Várias linhas de conexão 

Desvantagens: mais complexa que a serial, mais cara, mais rápida, altamente confiável, usado em dispositivos mais velhos ex: discos

2) Dispositivo de Bloco (Block Device)

↳ Bloco de tamanho fixo, cada um com um endereço de tamanho 128 a 1024 bytes

↳ Transfere um ou mais Blocos

↳ Referência de localidade → apelido probabilístico de se quer o 10 já pegou 9, 11, 12

↳ é mais otimizado ex's: HD, CD-ROM, Drive USB

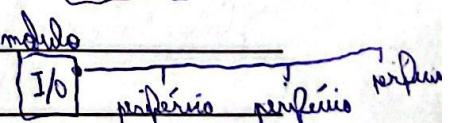
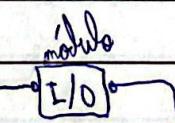
• Dispositivos de caractere (Character Device)

↳ Geram um fluxo de caracteres (não tem bloco, nem endereço, não divide pedaços reais)

↳ ex: arim em impressoras, mouse e interfaces de rede

3) Ponto a Ponto

↳ conexão direta, mais confiável, paralelo



• Multiponto

↳ não tem paralelo, mais confiável

Princípios de Hardware

↳ Componente Memória

→ (controlador do dispositivo \Rightarrow parte programável que controla o dispositivo é programada) e SO fala com o controlador



Ex.: Controlador de Disco, recebe um fluxo de bits com:

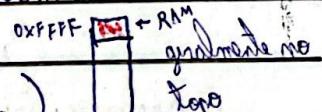
[preâmbulo | os bits do setor | Checksum (Error-Correcting-Code = ECC)]

Entende que está recebendo uma instrução, mete o byte em bloco, põe desem em um buffer interno, verifica o checksum e só copia o bloco para a RAM.

* Cada controlador possui registradores, CPU se comunica com ele através do:

↳ Mapeamento na memória

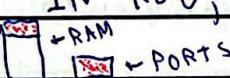
(registradores do controlador são tratados como endereços na memória)



↳ Controlador recebe um número de portas de E/S

(não é instrução especial) IN REG, PORT } OUT PORT, REG }

↳ Híbrido: os dois



Princípios de E/S (I)

Princípios do Software

↳ não deve se preocupar com os dispositivos (ex: read(), write() independente se é SSD, HD, ...)

↳ SO encapsula a complexidade dos dispositivos

↑ Erros: o mais perto possível do hardware.

modo de operação de E/S

1) E/S programada

2) E/S via interrupção

3) E/S via acesso direto na memória

1) Trabalho é feito pelo CPU (dados trocados entre SO e o módulo de E/S)

ex: a CPU executa um programa que: verifica estado do módulo (on/off ou não)

... envia o comando da operação, aguarda o resultado, efetua a transferência para o registrador da CPU.

FORONI

Desvantagem = CPU fica ocupada todo tempo (esperando em espera a resposta) até o dado ser dado, verifica se está certo, ai termina.

2) CPU recebe o sinal elétrico sendo avisado que um evento ocorreu.

Ele fica livre fazendo outras coisas até o dado chegar (interrupção) na CPU. Pq ele estávam no periférico. Esse tipo de interrupção tem um nível, quanto menor, maior é sua prioridade. Tem um controlador também.

3) Gravés do DMA (Directory Memory Access) → faz o papel da CPU no processo de entrada e saída.

Deixa a CPU livre?

A DMA pega o conjunto de dados do dispositivo de E/S, põe na memória

6 parâmetros → endereço na memória para guardar o dado, qntd de bytes,

~~Largura de banda!~~ Desvantagem: Se a CPU estiver desocupada → ela poderia fazer isso mais rápido

Tem o DMA a arquitetura seria mais barata.

Vantagem: controladora DMA faz todo o trabalho, deixa a CPU livre.

Técnicas de E/S. II

O Software de E/S deve ser independente do dispositivo

↳ Fazer o escalonamento de E/S

↳ Processador, ajuste de velocidade, qntd de dados transferidos

↳ Cache de Dados (armazena em cache de dados frequentemente usados)

↳ Reportar erros, proteção, definir tamanho de bloco independente do dispositivo

* Todo driver deve possuir uma interface padronizada (ter READ, WRITE,...)

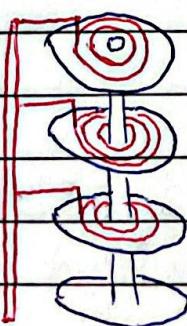
Mas cada driver de dispositivo é diferente, eles conversam com a controladora

do dispositivo, eles tem acesso ao SO → Fazem parte do Kernel

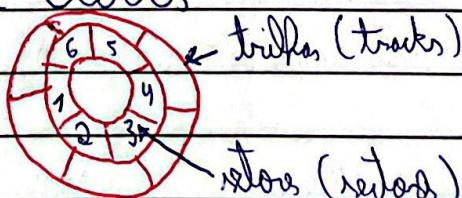
↳ print controller → printer drivers → SO

calcular move entre os cilindros

Disco e Discos



→ no cilindro



trilhas (tracks)

setores (sectors)

procurar: achar o cilindro → andar na trilha → procurar

FORONI Somado do disco: n° faces x n° trilhas x n° de setores x capacidade dentro do setor por setor.

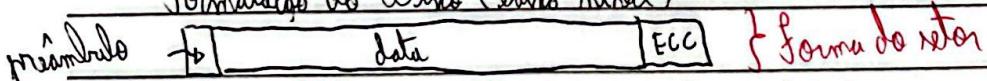
Seg Ter Qua Qui Sex Sáb Dom

Geometria do disco \Rightarrow Geometria virtual \Rightarrow encapsula a complexidade da geometria real.

O SO sabe que é tudo igual \Rightarrow x cilindros, y setores e z retas por trilha.

A controladora mapra isso pro real [discos modernos não em blocos]

Formatação do disco (livro nível)

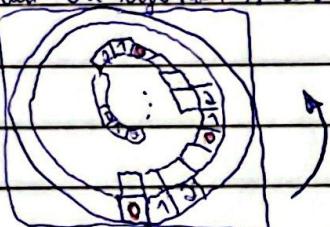


Próximo bloco = (fita que o setor está conseqüente) - Bloco de inicio - ID setor - ID cilindro

Buffon = data

ECC = Código de correção de erro

- Sons Cilíndricos ou cylinder stem (existem duas rotas completas quando troca de cilindro)



- Se o buffer for muito pequeno (controladoras antigas), e ele entra deixa, o tempo de escoviar o buffer para ler outra setor é o tempo de rodar por um setor, por isso intercalar é: $\frac{2}{13}$ \rightarrow maior tempo

Tempo de aceso: tempo de posicionar cabeçote em um determinado cilindro (trilha) + tempo de rotação (procurar o setor) + tempo de transferência p/ controladora

• Escalonamento do braço

FCFS (é quem movimenta muito longe o braço) 11 requisições 1, 36, 16, 34, 9, 12

SSF (requisição mais próxima) 11 requisições 1, 36, 16, 34, 9, 12 \rightarrow 12, 9, 16, 1, 34, 36

↳ move menos, mas tende a deixar o braço muito no centro (cilindro longe c/ inércia)

Elétrico (bit de direção UP \rightarrow prox requisição em sentido, DOWN \rightarrow muda o sentido)

11 requisições 1, 36, 16, 34, 9, 12 \rightarrow 12, 16, 34, 36, 9, 1

↳ RAID

* chegou na mesma sente o sentido

Raid (Redundant Array of Inexpensive Disks)

↳ armazena grandes quantidades de dados (Big Data), aumenta confiabilidade

O SO só vê 1 disco

Seg Ter Qua Qui Sex Sáb Dom

/ /
Clkts = Hardware (dispositivo que gera pulso, sincrono, síncr. para exec. de intruções)

frequência: n° de vezes que o pulso se propaga por segundo (Hz)

= Software → todo o resto é feito pelo clock driver (montar horário e data, controlar tempo total do processo, gerir uso da CPU (timeslice), alarm (sys call))

Prática

11

Linux é um UNIX - like

SEM ①

↳ dá pra fazer login por texto

↳ No terminal : `l \ f o " "` fala que o comando continua na
próx linhas

↳ ~ representa a área Home ~ usuários

↳ Variáveis de ambiente : PATH, HOME, SHELL

↳ dá pra exportar para programas

↳ uname - a (infos do sistema)

↳ find - name "filename" - print (acha e imprime o caminho)

(= grep name "filename")

↳ logout (sai do terminal)

• 1º processo lançado pelo Kernel do Linux : init process

↳ alias (formas atalhos para lista linhas de comando)

• As informações de inicialização do Linux (se for o inicio SO) ficam no MBR

↳ tail - n - 4 nome- arquivo (lista as 4 últimas linhas)

↳ cat my-file.txt (mostra o conteúdo)

↳ head my-file.txt X head - n 4 my-file.txt (mostra as 4 primeiras linhas)

• Começo de um script executável com : `#!/bin/bash`

SEM ②

System Calls : sysenter (intel) / syscall (AMD)

Wrapper functions → funções definidas em C para lidar com as "syscalls"

↳ retorno : 0 - OK

outro : é o valor absoluto do erro, geralmente é negativo. perror(3) ou strerror(3), só tá na variável errno (p) strerror_r(3) também

#include <errno.h>

errno

errno

Processo

→ #include <unistd.h>
 <sys/types.h>

→ duplio a partir do link seguinte

↳ pid - t pid = fork();

retorna o PID do filho

↳ faz um fork, duplio

O processo atual em processo pai: pid > 0 e filho pid == 0. Se der erro o pid < 0. → ai duplica o código, variáveis, stacks, ...

↳ G maioria usa o sistema copy-on-write, isto é, pai e filho recebem a mesma cópia da memória na leitura da imagem do pai. Se um deles deseja modificar essa imagem uma trap é feita e cada um recebe sua memória separada.

- Por isso não faz mais sentido usar o vfork(), nesse o filho compartilha as áreas de memória do pai, causando problemas.

- exit → encerra o processo .exit() em C. → fazendo um system call - exit().
- wait → faz o pai esperar o/algum filho → const EXIT_SUCCESS, EXIT_FAILURE
 ↳ wait(), waitpid(), waitid(), wait4()

↳ após o fim do processo filho, ele vêe com um gumbi até que o pai o recolha,
 (se o pai termina antes de pegar o filho, init adota ele ou o SO o mata.)

↳ Shell é um exemplo disto: → se for algo em 1º plano ele cria um filho para fazer e o pai (terminal) não faz nada até ele terminar. Mas se for algo em 2º plano ele não espera o filho.

Reflexão:

- Se a única maneira de criar um novo processo é duplicar o atual via "fork", como especificar um processo diferente especificado pelo usuário?

- execve() → pega o processo atual e substitui sua imagem pelo do processo passado.



Gerenciando Processo

prog ... cat // adiciona um programa para execução em 1º plano

prog & ... <enter> // adiciona em 2º plano

<ctrl>c [termina execução]

<ctrl>z [para execução]

FORONI

\$ ps = lista os processos
Seq Tid Qua Qui Sex Sét Dom

TRABALHOS / processos

/ /

\$ jobs = mostra processos iniciados na sessão shell (parado ou em execução)

\$ bg = envia para execução um job parado / suspenso ^{en background} bg % 3 → job #3

\$ fg = " " " em foreground " " " fg % 2 → job #2

\$ Kill PID → envia por padrão o SIGTERM

\$ Kill -l → lista os envios de sinal disponíveis (9 mata processo)

\$ Kill -X -PID → ^{envia um signal n'} ~~mata~~ o processo (9 mata)

\$ Killall → envia um signal pro processo pelo nome ex: Killall python -9

\$ pkill → envia signal processo por diferentes atributos.

\$ pgkill → " " para uma thread

\$ stty : exibe as configurações do console.

ro root pode mudar

PC B: Process Control

Processos tem prioridade de -20 → 0 → 19

\$ nice -n +10 prog significa executar processo de baixa prioridade

• Nos scripts: sleep suspende temporariamente a execução de um processo.

\$ time python // controlar o tempo de execução do processo

\$ top // mostra os processos com mais ocupação da CPU

a) para processo em foreground: <ctrl> C

b) " " " " background: fg; <ctrl> C ?

c) " em outra sessão:

ps -ef | grep [nome-do-prog ou "login"]; Kill -9 pid

ps -U login; Kill -9 pid

Killall -9 nome-do-prog

• Redirecionamento de E/S de processos 1.8 (Processo → Sem 2)

Threads

O select() tem a mesma função da chamada fselect com parâmetro F_SETFL e atributo O_NONBLOCK

FORONI

Chamada de sistema + `clone()` especifica para criação de threads, não pode criar um novo processo ou só um mix entre ambos.

↳ faz isso passando vários flags do que será compartilhado entre a criada e a criadora. → aí dá pra indicar os mesmos que uma thread compartilha

→ thread

Flags = CLONE_VM | CLONE_FS | CLONE_FILES | ...

No linux tudo é tratado com tarefas, logo, o que diferencia uma thread de um processo é o que ela compartilha/ou tem copiado.

Threads ↗

→ processos

Compilar para threads:

`gcc programa.c -lpthread -o programa`

Semana 3

Espera de tempo.

Comando nice : mudar a prioridade dos processos

Escalarador Linux é baseado em time-sharing

escalador é
muito

Prioridade estática : usado por processos de tempo real, pode ser definido pelo user e o escalador não

Prioridade dinâmica : Alterada, calculada pela quantidade de tempo restante em seu quantum

#include <sys/time.h>, <sys/resource.h>, <stdio.h>, <unistd.h>

int setpriority (int which, int who); → 0 se foi, -1 se deu erro

int setpriority (int which, int who, int prio); → prioridade -1 se não deu certo

PRIO_PROCESS {getpid()} {a prioridade nova}

Deadline

Afinalidade do processador: probabilidade de um processo ser executado de forma consistente no mesmo processador.

FORONI

~~shed~~
sched_getaffinity ()

CPU_ISSET (i, my_set) // se a CPU estiver no bit map.

↳ cpu_set_t my_set; // define uma máscara de bits da CPU

CPU_ZERO (& my_set); // iniciado com 0, nenhum CPU selecionado

CPU_SET (7, & my_set); // set o bit map para representar o core 7

↳ retorna -1 em caso de erro no bit map (não tem a CPU)

↳ sched_getaffinity (0, sizeof (cpu_set_t), & my_set);

↳ set a máscara desse processo para a máscara, onde caso o core 7, o núcleo desse processo
não rodará no core 7.

↳ sched_getaffinity (0, sizeof (cpu_set_t), & myset)

↳ retorna colocado no bit map ou -1 em caso de erro.

↳ sched_getcpu () → retorna a CPU em que o processo está rodando

↳ politicas de escalonamento com filhos multithreaded e timesharing

• SCHED_OTHER = política circular padrão com prioridade

• SCHED_IDLE = executar tarefas em segundo plano → prioridade baixa

• SCHED_BATCH = "processo no estilo lote"

• SCHED_FIFO = first in first out

• SCHED_RR = round robin classificado circular

↳ tipo: pid_t

↳ int policy = sched_getscheduler (pid)

↳ selmanu 4

• Usa-se nvin para a comunicação entre processos especialmente entre pai e filho.

↳ Pode-se usar o comando kill (2) para enviar sinal para outro processo do mesmo usuário ou do root.

CTRL+C (SIGINT)

0

FORONI

- Reliable Signals (nível normal): eventos pré-definidos, números 1 a 31.
- Real-time Signals (níveis normais): valores definidos pelas Macros SIGRTMIN (33) e SIGRTMAX (64), podem ser gerados de acordo com a lógica do programa que os utiliza.

- ↳ Cada sinal é composta de um grupo de regras:
- Term: "termina" → termina o processo
 - Ign: "ignora" → ignora o sinal
 - Core: "termina" → gera um dump core
 - Stop: "para" → atinge o processo para pausá-lo
 - Cont: "continua" → continua o processo
- tem uma lista no SEM4
↓ que dizem com o sinal
- processos possuem estruturas para capturar, ignorar, tratar ou bloquear sinais. São os níveis SIGKILL e SIGSTOP não podem ser tratados.
 - raise(3) → sinal para thread corrente ou sequência (3)
 - pause(2) → bloquem o processo ou thread atual à espera de um sinal.
 - signalpend(2) → " específico.
 - Excepto SIGKILL e SIGSTOP, todos os outros níveis ficam pendentes até o processo ser executado.
 - ↳ um sinal também pode matar um processo (é intuito específico)

SINAIS (#include <signal.h>)

- signal → implementa uma função de tratamento ou que ignora


```
if (signal (int signum, signalhandler handler) == SIG_ERR) {  
    signal a ser tratado  função prototipada  
    ..  
}
```

→ retorna: o endereço da função "anteriormente tratada"
- signal(SIGACTION) → tratar ou ignorar um sinal; permite bloquear o recebimento durante o tratamento também e salva o estado anterior do tratamento.

- Kill → envio de sinal para processo definido pelo seu pid


```
kill (pid, SIGTERM)  
    ^ pid          ^ ex. de sinal.
```

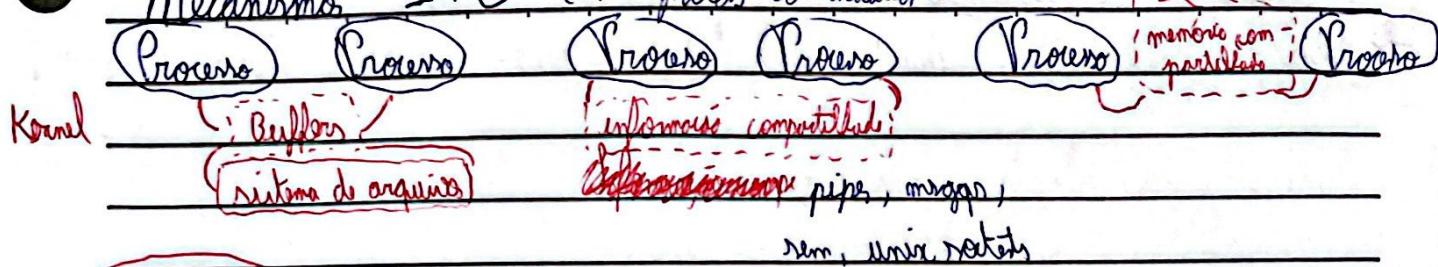
Têm vários níveis de interesse: Demanda 4, 1.2 Tratamento e encaminhando sinais.

→ Até que encontre nível anim: \$ kill -> SIGINT PID

FORONI

SIGHUB

Mecanismos - IPC (Intercâmbio de comunicação)



Pipes e FIFOs

• Usados para comunicação bidirecional entre processos (é um canal, que um envia e outro le) → entre pais e filhos

• ñ necessariamente entre pais e filhos implementação em msgpipe.c

• Filas de mensagens: troca de bytes entre processos. (lista de mensagens armazenadas no Kernel) → um processo copia a mensagem e manda para o SO, que pode salvá-la no Kernel-space e ai outro processo no user-space pode pegá-la copiá-la para o user-space e le-las.

• dá prioridade entre as mensagens. → código writer.c e reader.c

• Semáforos → estrutura para sincronização de atividades entre processos → podem ocorrer consultas/alterações ao valor de um "contador" associado ao semáforo

• ter → o contador é uma fila de processos bloqueados

• Os semáforos SysV têm 2 filas (para os que querem decrementar o contador e para os que esperam de fato 0).

• Podem ser: Binários (para fazer exclusão mútua) ou com contadores (conta reverso).

• Sistemas Semáforos SysV e POSIX → dão para controlar múltiplos recursos simultaneamente.

#include <sys/types.h> → faz um array de semáforos contadores

→ semget (Key_t Key, int num-sem, int sem-flags) → cria ou obtém um identificador existente
um valor intiero para permitir q processos acessem o mesmo sem número de semáforo modo de criação
normalmente: 1 → IPC-CREATE → só processo que crie usa

→ retorna ipc_id, identificador do semáforo

→ " -1 se erro

IPC-PRIVATE → cria o semáforo se n existe
IPC-EXCL → se n existe o semáforo existe

FORONI

-1 deu ruim / 0 deu bom

~~semop()~~

→ semop(semid, struct sembuf *rops, nsems, nrops) // modifica o valor do semáforo

Identificada de semáforo retornado ponteiro para uma matriz m de operações dessa matriz
por semget() de operações o Nsem executadas no semáforo

struct sembuf { semnum; /* valor do semáforo */ int semop; /* operação do semáforo */ int sem_flg; /* flags da operação */ }

Ex: struct sembuf rem-lock = { 0, -1, IPC_NOWAIT } // bota no momento ou falta antes

if((semop(rem_id, &rem-lock, 1) == -1) error("semop"))

struct sembuf rem-lock = { 0, 1, IPC_NOWAIT }

→ semctl(semid, semnum, cmd, union semun arg)

identificadores

valor do sem (0)

argu. adicionais

{IPC_STAT & IPC_SET}

SETVAL → inicia o semáforo com valor contido (val union semun)

IPC_RMID

Todos programas com pthread: gcc nome.c -lpthread -o nome.exe

Seg Ter Qua Qui Sex Sáb Dom

Semana 5 → Sincronização de Threads

② Detach e join:

- + pthread_join: espera o fim de uma thread (tipo wait / waitpid)
 - ↳ código disponível → meu_join.c ou Sync_Threads_pt-join.c.
- + pthread_tryjoin_np() → é não bloqueante, só retorna um erro se ainda não terminou
- + pthread_timejoin_np() → especifica um tempo, se dps desse tempo a thread não terminou ele retorna um erro
- + pthread_detach() → muda o status de uma thread para detached

③ Semáforos POSIX: semáforos são contadores de recursos compartilhados entre threads

#include <semaphore.h>

- ↳ sincronização de maneira bloquante ou não
- ↳ operações: incrementar o contador de maneira átomica
 - esperar até que o contador seja ≠ de 0 e decrescê-lo de maneira átomica

// global
sem_t sem;

- sem_init(&sem, 1);
↳ inicia um semáforo compartilhado → número de recursos
- sem_post(&sem);
↳ libera um semáforo → incremento (up) → adiciona um recurso
- sem_wait(&sem);
↳ espera até que seja liberado → decremento (down)
- sem_timedwait(&sem, ...);
↳ similar ao wait só que espera no tempo

FORONI

+ * rem - t * rem - open (const char * name , int oflag ...)
 ↳ abre um nome remplace com a flag OFLAG

+ ↳ int rem - unlink (const char * name)

↳ remove um nome remplace NAME

+ int rem - close (rem - t * rem)

↳ fecha o descriptor de um nome remplace rem

④ Mutexes (mutual exclusion) → protege regiões críticas, garante a exclusão mútua.
 pthread - mutex - t lock; // global

+ if (pthread - mutex - init (& lock , NULL) != 0)
 ↳ inicia um mutex e retorna 0

+ pthread - mutex - lock (& lock); → fecha região crítica / thread é bloqueada se já bloqueada

+ " " - unlock (& lock); → abre região crítica

+ " " - destroy (& lock); → destrói o mutex

+ " " - timed lock (pthread - mutex - t * - rettent - mutex)

↳ tenta bloquear em mutex, se já tiver bloqueado bloqueia a thread
 até ser liberado ou o timeout especificado expirar

mutex.c
 ...
 ↳ ocorre: erro ETIMEDOUT é retornado

④ Read / Write locks - rwlocks → permite o bloqueio relativo entre leitores e escritores

- Bloqueio de leitura: impedem novo escritor, mas não a leitor

- " " escrita: " " a escritor e leitores.

+ int pthread - rwlock - init (pthread - rwlock - t * retstart_rwlock , abrlck);

+ " " - rwlock (" " * rwlock), → bloqueio de leitura

+ " " - wrlock (" " * rwlock); → bloqueio de escrita

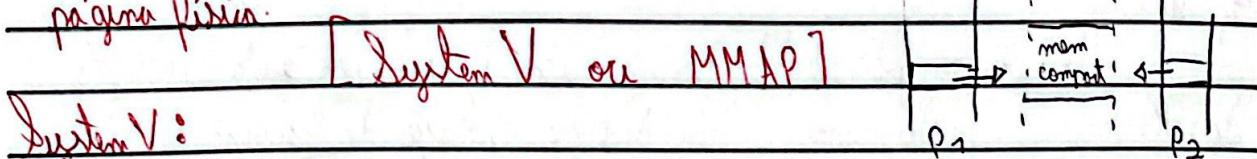
+ " " - unlock (" " * rwlock); → desbloqueio

④ Barreiros

~~FORONI~~ * material escrito

permite o sincronismo de threads em ... ponto da execução.

- ② Memória Compartilhada → compartilhar memória entre processos é a mais eficiente no IPC
 ↳ é obtido fazendo com que páginas lógicas apontem para a mesma página física.



System V:

1º chamada ao SO para decorar de um espaço na memória

`shmid = shmat(...)` ← `shmat.c` → flags IPC-CREATE

2º acoplar essa área de memória (fazer aponta para da

`if(mem = shmat(...))` ← `shmat.c` shmat → (se nenhuma existe, cria)
com → (já existe)

// dí pro exercícios com `streqy(mem, MEG);` por exemplo; SHM-R → read access
SHM-W → write access for user

3º acoplar essa área de memória a outro processo

`if(mem = shmat(...)) == P-1)` ← `shmat2.c`

`if (int shmdt(shmid, cmd, &buf) == -1) { error}`

`shmdt.c` IPC-STAT → prender os shmaters buf, → com info sobre
IPC-RMID → remove a área do shmid.

4º int shmat (cont void *shaddr) → desacopla o segmento criado por shmid()

↳ o conteúdo é desapto? endereço obtido por shmat()

Seteira 6

- ③ Barreira: Determina um ponto de execução em que os threads são bloqueados enquanto os outros não chegam até lá; quando todos chegam, os threads saem;
- `pthread-barrier-t barrier;` contador de threads
- `pthread-barrier-init(&barrier, NULL, N+1);` esperando.
- `pthread-barrier-wait(&barrier);` fez uma thread esperar a barreira → contador +1
- ↳ Como usar:

Barreira/ex 1-barrier.c

① Produtos e Consumidores

Seg Ter Qua Qui Sex Sáb Dom

↳ Operações Critônicas são operações que não podem ser interrompidas

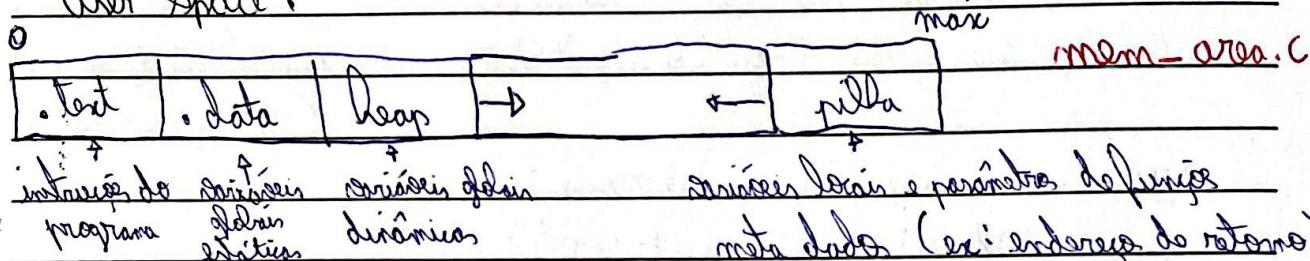
Exemplo:

Uma forma mais eficiente de sincronizar threads sem muito movimento do Kernel, é usado como a base de outras soluções de exclusão mutua.

Semana 10

Gerenciamento de Memória

User Space:



ASLR

Address Space Layout Randomization → coloca em locais diferentes a pilha toda vez que o processo executa → segurança

• Tem mais utilitários (já no código de compilar) para manter o código memória

Alocagem de variáveis:

• Estática: Continua com o mesmo espaço definido na compilação até o fim do programa.

DATA = variáveis inicializadas ou BSS: não inicializadas

• Dinâmica: espaço para armazenar as variáveis de funções é alocado quando a função é invocada e liberado quando ela termina na pilha (STACK) Dem pro seu uso.

• Dinâmica: malloc(), free(), new(), ...

usa a área Heap delimitada pelo ponteiro PROGRAM-BREAK

XXXX <lib.2>

malloc = número de bytes

calloc = (número, size bytes) → tam de cada elemento → inicia com 0's

free() = libera o espaço liberado por ptr

Kmalloc (size, flags) = aloca de uma flag

FORONI

Seg Ter Qua Qui Sex Sáb Dom

- Fragmentação de memória: de tanto usar o mello a memória fica fragmentada
→ memory leak: não liberam a memória depois, como toda a memória

↳ Analisa iso:
- splint: sudo apt-get install splint [estático]
- valgrind: sudo apt-get install valgrind [dinâmico]
ex: splint memory-leak.c [vê o código]
valgrind ./memory-leak [é de debug → tempo de execução]

• Políticas de alocação de memória

- first fit - best fit
- Next fit - worst fit

↳ Dendrois o tipo de política de alocação de memória do mello.
ver o arquivo fit.c

(com base nos: first-fit.c, worst-fit.c, best-fit.c, next-fit.c)

↳ falou o como depois a execução; o meu SO, pôr os links de execução

Semana 11 → Memória Virtual

Semana 12 → Cargando

① /proc → quando as informações sobre o SO.

\$ stat arquivo/folder ⇒ mostra info do arquivo.

ou programa: stat.c → põe os dados do arquivo em
ou lstat.c uma struct e dia modificar
ou fstat.c // sempre dia certo?

dis.c // faz a leitura de diretórios

Istreams → File Descriptos

funções mais de alto nível → system calls → constituem uma interface de baixo
"para executar entrada e saída" read(), write(), fseek(), ... nível
não facilitado

int fprintf(FILE *stream, const char *...); fprintf.c | fprintf2.c

int fscanf(...)

FILE fopen(...)
int fgetc(...)

FORONI

lseek - write.c → mostra 2 formas de escrever bloco em arquivo
 ↳ escreve 'K' na posição 10 ⇒ lseek - pulrite K 10

fdisk.c

↳ fdisk = entrada padrão (leitura - 0)

↳ fdout = saída padrão (escrita - 1) ↳ filero.c

↳ fderror = saída de erro (escrita - 2)

3 portadas ↳

~~~~~ Primitiva (chamadas de sistema)

→ open() e creat() ⇒ abre ou cria / cria um arquivo

↳ flags: O\_RDONLY (não leitura), O\_WRONLY (não escrito) ...

→ ssize\_t write(int fd, const void \*buf, size\_t nbyte) ⇒ escreve em um arquivo O

↳ escreve no buffer do kernel especificado

Kernel primeiro

→ « read (000, size\_t count) ⇒ lê o que tá no buffer.

[cria.c] → só sobrepõe no arquivo

[append.c] → escreve no final do arquivo open(args[1], O\_RDWR) APPEND

↑  
por causa  
dito

→ dup() - dup2() ⇒ redirecionam a entrada - redireciona a saída

Semana 13

↳ Partição no Linux = Devmap [mata para operar interface virtual]

↳ Proc [sistemas de arquivo do Kernel]

sudo fdisk -l

ls -l (mostra os permisos)

ls -i (nome-do-arquivo) ⇒ nº o i-node

ls -li

## XV6

- núcleo monolítico
- memória (memória e Kernel)
- processos (system call → execução entra no espaço do Kernel, redige o endereço e volta<sup>1º</sup> espaço de usuário)

- Sempre que fizemos uma modificação:

\$ make clean → make

\$ make gmake

- Para adicionar programas ao XV6:

→ colocar o programa.c dentro da pasta do XV6

→ Alterar o Makefile em: PROGS

  \* EXTRA , para reconhecê-lo

→ recompilar

↳ make clean

make

⇒ depois é só executar: teste

make gmake

Exercícios chamada de sistema