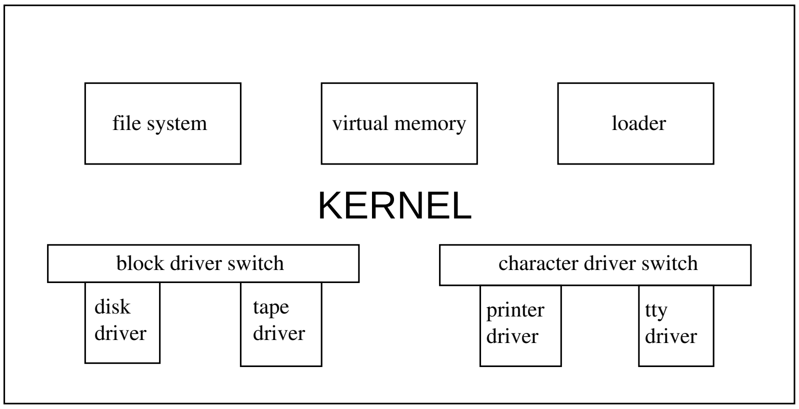
Sistemas Operacionais

Vítor Santos Costa
COPPE/Sistemas

O Kernel Unix

- Inicialmente: “small is beautiful´’.



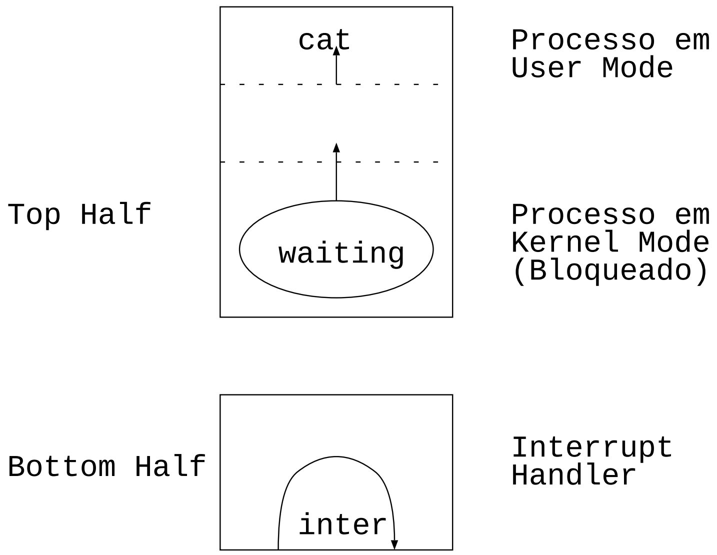
- Versatilidade é necessária: múltiplos sistemas de arquivos, múltiplos formatos de execução.

Alocação de Memória

- Em Unix, um processo executa em 2 modos:
  - Modo Kernel
  - Modo User
- Cada processo tem memória virtual:
  - kernel ou system space: partilhada por todos os processos.
  - memória privada: (pilha, bss).
  - memória partilhada: código, bibliotecas partilhadas, mmap
  - Privadas mas controladas pelo kernel: *área-u* e *pilha do kernel*.

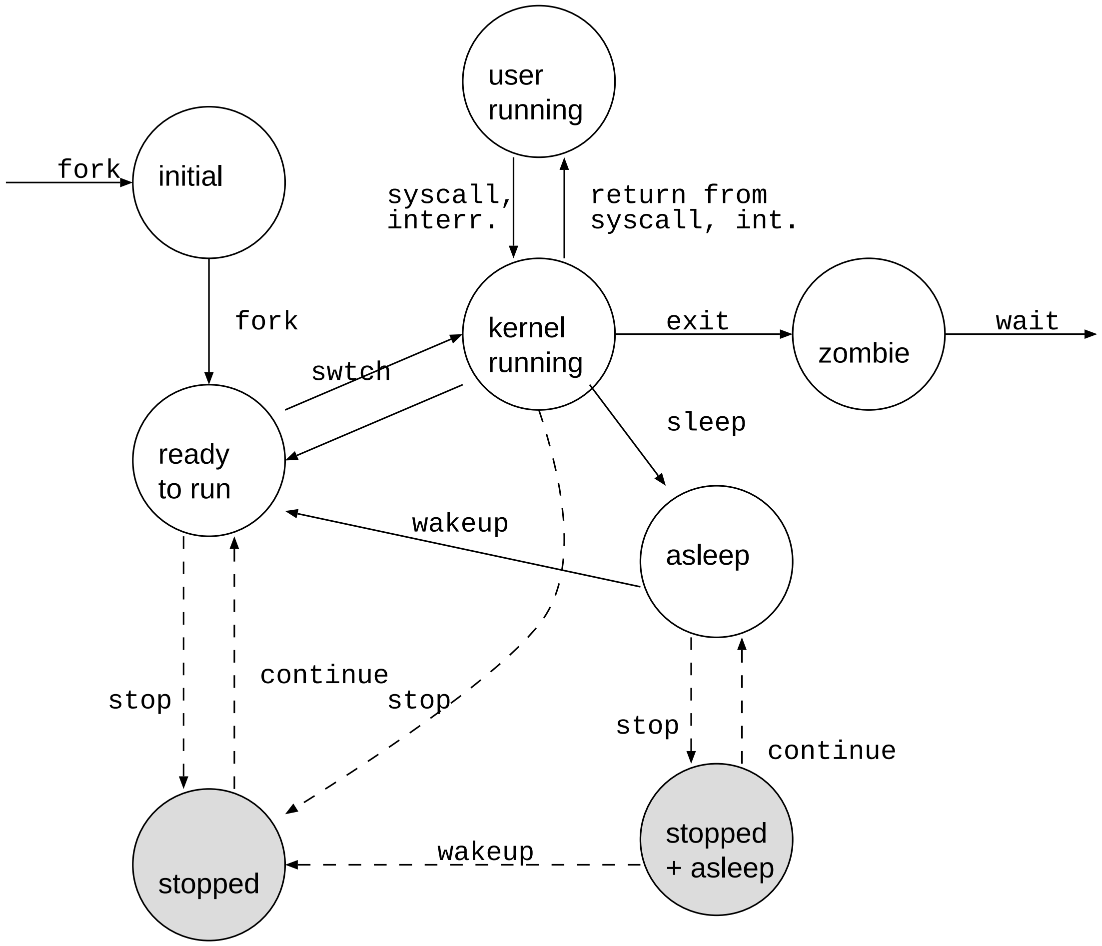
Modos de Execução e Contextos

- Funções do Kernel podem executar no contexto:
  - Processo (ie, syscall): kernel pode bloquear.
  - Kernel, ou interrupt: não pode bloquear, usa kernel stack.



- Top-half e bottom-half do kernel.

Estados de um Processo



Contexto de um Processo

- Espaço de Endereçamento do Usuário: texto; dados; pilha do usuário; memória partilhada.
- Informação de controle: área-u; proc; pilha do kernel; mapa de tradução de endereços.
- Credenciais: UID, GID.
- Variáveis de Ambiente
- Contexto HW: PC, SP, PSW (processor status word), mmem regs, fpu regs.

Credenciais de um Processo

- UID/GID real e efectiva.
- Efectiva: usada para abrir ficheiros.
- Real: usada para enviar sinais.
- programs em suid mode: mudam UID efectivo.
- programs em sgid mode: mudam GID efectivo.
- `setuid()` ou `setgid()`: permitem voltar ao ID real.
- BSD suporta vários grupos por utilizador.
- SYSV mantém saved UID e GID que sao restaurados por setuid.

A Área-U

- PCB (process control block): armazena o contexto HW quando o processo não está activo.
- pointer to `proc struct`; pilha do kernel.
- UID e GID real e efectivo
- argumentos e resultado da syscall corrente.
- signal handlers.
- info sobre texto, dados e pilha, mais gestão de memória.
- FD abertos (dinâmico ou estático).
- nó-v do directório currente e do terminal currente.
- estatísticas (CPU, profiling, quota).

A Estrutura Proc

- PID e SID (id da sessão).
- endereço da área u.
- estado do processo.
- ptrs para incluir o processo numa fila de escalonamento ou de “sleep”.
- “sleep channel”.
- prioridade de escalonamento.
- sinais que são aceites pelo processo.
- gestão de memória.
- ptr. para lista de processos activos, livres ou zombie.
- Ptrs. para hierarquia de processos e para hash queue on PID.
- Misc.

Execução em Modo Kernel

- syscall:

  Um wrapper chama a instrução `chmk` no VAX, `syscall` no MIPS, `trap` no MC68k, ...

  `syscall()` no kernel copia arguments e salta através da tabela `sysent`.

  No retorno copia valores de retorno em registo, restaura contexto HW, e regressa a user mode.

- Interrupt Handling

  Vários Interrupts: *interrupt priority level (ipl)*: 0-31 em BSD.

  Quando saímos do Interrupt Handler verificamos se há alguma interrupção suspenso.

  Se o nosso nível for $<$ que o nível corrente, guardamos interrupção num registo especial.

Sincronização em Unix

- syscall são "non-preemptive": só um processo de cada vez.
- Processo pode bloquear num recurso (ie. buffer em memoria):
  - Processo chama `sleep()`.
  - `sleep()` coloca processo numa fila, e chama `swtch()` para entrar outro processo.
  - quando o kernel liberta recurso chama `wakeup()` para acordar todos os processos, ie, colocá-los na fila do escalonador.
  - outros processos podem precisar do recurso: `wake_one()` e `wakeprocs()`.
- Interrupts podem acontecer a qualquer altura: sinc. com `splbio` and `slpx`.
- Mais complicado ainda com multiprocessamento.

Sinais em Unix

- Funcionalidade: comunicação entre processos, interrupts, e excepções.
- Cada sinal tem uma resposta default, geralmente terminação do processo.
- Problema: sinais sobre processos adormecidos:
  - se vai acordar cedo, o sinal pode bloquear.
  - se vai ficar muito tempo, pode-se passar o interrupt.

  BSD fornece `siginterrupt()` para controlar esta funcionalidade.

Inicialização de Processos: `fork`

A chamada `fork()` duplica processos:

- `fork()` cria um novo processo: retorna 0 para filho e PID para o pai.
- `fork()` reserva swap, aloca novo PID e proc, inicializa proc, aloca mapas de tradução de endereços, aloca u-area e copia do pai, altera a u-area com novos mapas de endereços e swap, adiciona o filho aos processos que partilham o texto do pai, duplica as áreas de pilhas e dados do pai, obtém referencias a recursos partilhados, inicializa contexto HW, põe o processo runnable e na filha de escalonamento, retorna para o filho e para o pai.
- evitar cópia: copy-on-write e `vfork()`.

Inicialização de Processos: `exec`

A execução de um novo programa é iniciada com `exec()`:

- `exec()` faz overlay com um novo programa.
- `exec()` obtém o executável, verifica permissões, lê o cabeçalho, altera se SUID ou SGID, coia os argumentos de `exec()` e *env* para kernel space, aloca swap para data e swap, liberta antigos data and stack, aloca mapas de endereço e inicializa-o, restaura *env* e argumentos, reinicializa os signal handlers, inicializa contexto HW.
- entre os dois podemos fazer coisas como alterar os descriptores de entrada-saida, etc.

Espaço de Execução do Processo

Texto código executável.

Dados Inicializados Objectos explictamente inicializados.

Dados Não Inicializados (bss)

Memória Partilhada

Bibliotecas Partilhadas

Heap

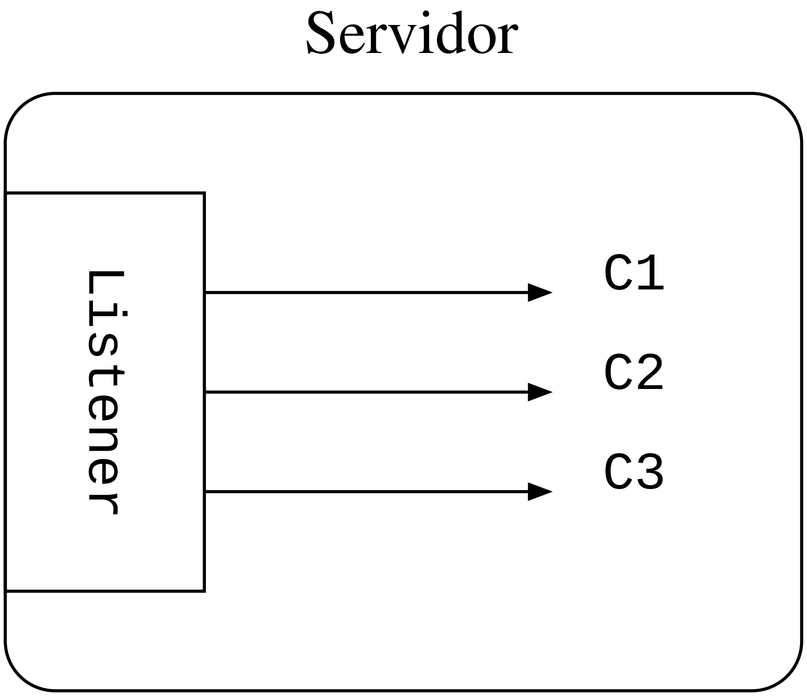
Pilha do Utilizador

Terminação do Processo

- `exit()` é chamada ou de um signal ou do próprio processo.
- `exit()` desliga sinais, fecha ficheiros, liberta ficheiro texto e outros recursos como cwd, escreve no log, guarda estatísticas, muda para `SZOMB`, faz com que *init* herde o processo, liberta memória, envia `SIGCHLD` para pai, acorda pai, chama `swtch()`
- `wait()` espera terminação de processos.
- se processo morre antes do pai e este não chama `wait`, processo fica *zombie*. SVR4 permite usar `SA_NOCLDWAIT` para indicar que pai não vaiu esperar pelos filhos

Porquê Threads

- Aplicações cliente-servidor



- Aplicações Paralelas
- Aplicações Interactivas
- Problemas:
  - Preço de `fork`.
  - Partilha de Recursos, como memória.
- Solução: *threads* dentro do mesmo processo.

Concorrência em Processos

- Concorrência de Sistema: o kernel reconhece múltiplos *threads* dentro de um processo.
- Concorrência para Utilizador: independente do kernel, util para aplicações concorrentes.
- Concorrência Dual: kernel reconhece múltiplos threads num processo, e utilizador pode usar libraria para definir os seus threads.

Tipos de Threads

- Kernel Threads: baratos, não são associado com processos de utilizador, e têm a sua própria pilha.

  Úteis para AIO e interrupts. Equivalente ao *pagedaeomn* e a *nfsd* em Unix.

- Lightweight Process: kernel supported user-thread.

  Podem fazer syscalls e bloquear.

  Podem correr em CPUs diferentes.

  Precisam de mais estado do que KT: pilha e contexto de registos. Precisam de syscalls para serem criados, sincronização (para blocking), context switching e escalonamento. Isso obriga a 2 *mode switches* atravessando *protection boundary*.

- User Threads: implementados por bibliotecas (C-threads ou pthreads).

  Interação é rápida, escalonamento é feito pela aplicação. AIO permite não bloquear thread à custa de complexidade de programação. Ideais para window systems.

Problema: separação entre UT e LWP: kernel não pode saber que LWP tem os melhores threads; UTs podem perder LWPs, não suportam paralelismo.

Suporte a LWPs no Kernel

- suporte a `fork()`: duplicar todos os LWPs ou apenas o que fez `fork()`?

  1. Melhor para `exec()`. problemas com bib. que tenham seus próprios LWPs (as suas estruturas de dados não vão ter LWPs).
  2. LWPs bloqueados? Possível LWP retornar `EINTR`, mas tem que se ter cuidado com fechar ligações de rede. Cuidado com estruturas de dados externas.

  Registrar *fork handler* que são executados antes e depois de `fork()`.

- Vários LWPs podem aceder ao mesmo fd (um lê, outro faz fseek). Soluções: aplicação resolve o problema, ou kernel suporta random IO atómico.

  Gestão de cwd, credenciais, e mapa de memória (`brk()` ao mesmo tempo).

- Quem recebe sinais? Todos os LWPs (Z), um arbitráriamente, um master, heuristicas, novo LWP.

  Sinais como SIGSEGV devem interromper LWP responsavel. SIGINT é complicado.

  Signal Handler geral ou privado? Signal masks devem ser privados.

- Visibilidade fora do processo? Dentro do processo (signal entre LWP)?

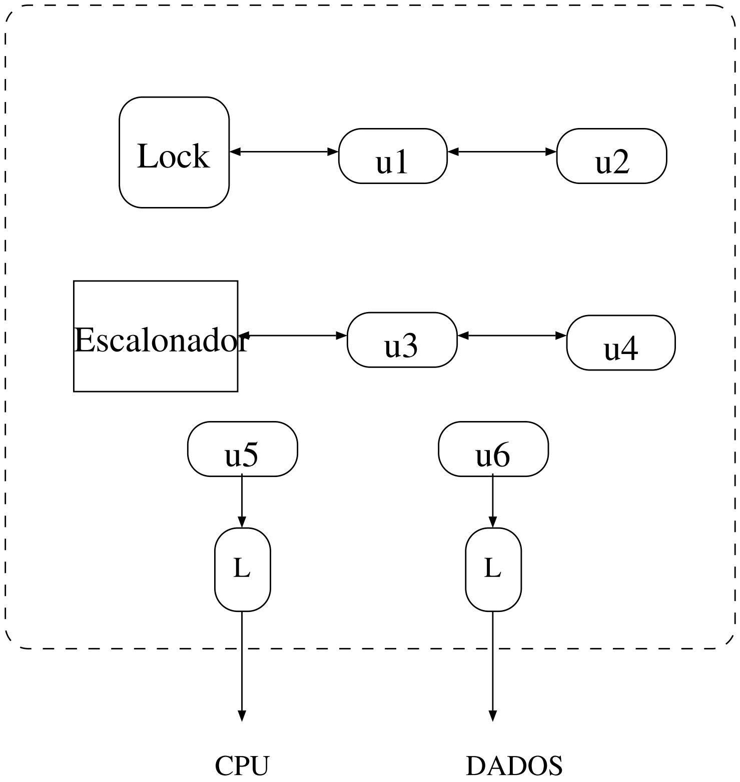
- Gestão das pilhas: o kernel não pode saber.

Bibliotecas de Threads

- Interface
  - criar e terminar
  - suspender e reexecutar;
  - indicar prioridades
  - escalonar e mudar contexto
  - sincronização (semáforos, mutexes).
  - enviar mensagens entre threads.

Implementação de Bibliotecas

- Implementação
  - LWP para UT
  - Multiple UT em LWP
  - Permita UT ligados e não ligados. Pode favorecer bound threads



Kernel Level Threads

Kernel Threads são usados para actividade assíncrona e para suportar LWPs:

- Cópia dos registos
- Informação sobre prioridade e escalonamento
- ptr. para lista de escalonamento ou lista de suspensão.
- ptr. para pilha
- ptr. para LWP e proc se associado a WLP, + info sobre LWP.
- ptr. para fila de threads no processo e no sistema.

Um dos objectivos é impedir *inversão de prioridades*.

Solaris: LWPs

Cada LWP é associado a um KT durante a sua vida.

Em `lwp`:

- valores de registos
- argumentos e resultados para `syscalls`.
- info. para signals.
- uso de recursos.
- alarmes em tempo virtual; tempo de util. e CPU.
- apt. para KT.
- apt. para `proc`

`lwp` é swappable, então máscaras têm que estar em KT. No SPARC **g7** refere `lwp`.

Sincronização por mutex, variáveis de condição, semáforos, rw locks, bloqueantes ou não.

Signal handlers são comuns ao processo mas máscaras pertencem ao LWP (e possivelmente stack).

Solaris: UTs

UT são implementados por bibliotecas. Podem ser associados a LWPs ou não

- Thread ID.
- saved register state
- pilha de utilizador.
- máscara de sinais.
- prioridade
- armazenamento local (`errno`).

Solaris: Interrupts

Solaris usa mutex e semáforos para interrupts, e usa *interrupt threads* para atender a interrupts:

- evitar o custo de KT: pool de threads, inicializados parcialmente. Um por cada nível exigindo 8k de espaço.
- Int. threads operam sem estarem completamente inicializados. Só são inicializados se bloquearem.
- Durante a sua execução prendem o thread que foi interrompido.

`fork` duplica todos os LWPs (os em `syscall` recebem `EINTR`). `fork1` só um thread. `pread` e `pwrite` fazem `fseek+op`.

Mach: Princípios

Mach suporta:

- Task: objecto com espaço de endereçamento e recursos chamados *port rights*.
- Thread: é a unidade de execução com kernel stack, estado, e escalonável. Se do kernel pertencem à *kernel task*.
- `syscalls` manipulam tasks e threads (create, terminate, suspend, resume, thread_status, thread_mutate e task_threads).
- Biblioteca *cthread* fornece interface. Pode ser com `coroutine`, precisa de `cthread_yield()`; `threads`, ou `tasks`.

Mach: Continuações

Problema:

- Cada thread tem pilha com pelo menos 4KB: overhead.
- Unix usa modelo de processos: cada thread tem uma pilha e pode bloquear sem salvar a sua pilha
- Modelo de interrupts: uma única pilha do kernel, e o processo interrompido tem que salvar o seu estado.
- Primeiro modelo é melhor quando processo tem muito estado. Se tiver pouco estado, segundo é o melhor.

Mach: Continuações

Mach usa *continuações*, uma função a executar quando o thread bloqueia:

- Thread primeiro guarda algumas variáveis e devolve a sua pilha.
- Quando é reexecutado, usa variáveis e a continuação para recuperar estado.
- Exige conhecimento da rotina que bloqueia e da continuação.

Mach: Continuações

- Uteis quando há pouco estado (exemplo, no fim de page fault handling enquanto espera pelo resultado de um read.
- Se kernel novo e antigo têm continuações kernel pode transferir pilha directamente: evita TLB e cache misses.
- Se cliente chama `mach_msg()` e tem um servidor à espera, cliente pode passar-lhe a pilha e bloquear com `mach_msg_continue()`, que execute aimediatamente, e depois devolve a pilha.
- Ideal para hot-spots (típicos em interface pequena, como num micro-kernel).

Mach: Implementação

Escalonador efectua um *context switch*

- guarda registos em PCB (u-area).
- recupera registos de outro PCB.
- cache flush, se necessária, força mais acessos
- flush de instruction pipeline.

Clocks

Resposta a interrupt de clock deve ser o mais rápida possível:

- reiniciar hw, se necessário.
- incrementar estatísticas.
- recalcular prioridades e time-slice.
- enviar um `SIGXCPU` para processo se excedeu quota.
- alterar tempo real.
- processar *callouts*
- acordar processos de sistemas como *swapper* e *pageout*
- processar alarmes.

Callouts

Funções a chamar mais tarde (*timeout* ou *task queue*):

- Retransmissão de pacotes;
- Funções do escalonador e gestor de memória;
- Monitoração de devices;
- Polling

Interrupt handler coloca uma flag que é verificada no retorno à prioridade normal.

Callouts são ordenados por "tempo até disparar" em BSD, existem outros esquemas.

Alarmes

Alarmes são activados ao fim de um certo intervalo de tempo:

- Tempo-Real, `SIGALRM`
- profiling, `SIGPROF`
- virtual-time, `SIGVTALRM`
- BSD usa `setitimer()`, microsegundos, mas funciona em ticks.
- SVR4 fornece `hrtsys()`
- Note que processo só responde ao sinal quando é escalonado, o que afecta precisão.

Política de Escalonamento em Unix

Três tipos de aplicações: interactivas, `batch` e real-time.

- Unix tradicional desenhado para aplicações interactivas.
- Cada processo tem uma prioridade que varia dinâmincamente.
- Processos de mais alta prioridade tiram outros processos do CPU mesmo quando o processo não terminou o seu quantum.
- Kernel é non-preemtible: processo só returna o CPU quando bloqueia ou quando regressa a User Mode.
- Prioridades: `0 a 49` para kernel, `50 a 127` para user-mode.
- Em `proc`: `p_pri`, prioridade corrente; `p_usrpri`, prioridade em modo utilizador, `p_cpu`, uso de CPU, e `p_nice`.

Algoritmo de Escalonamento em Unix

- Depois de bloquear, `pri` é associada à prioridade do recurso (eg., 28 para terminais e 20 para disco).
- quando regressa a user mode, volta a usrpri.
- `nice` pode ser usado para controlar prioridades.
- p_usrpri = PUSER + (p_cpu/4) + (2*p_nice)
- `p_cpu` decai por um factor de 1/2 em SVR3 e $(2 * load\_average)/(2 * load\_average + 1)$ em BSD, activado de segundo a segundo por um callout. $load\_average$ é o número médio de processos executáveis no último segundo.
- Prevenir "starvation". Factor dependente do load evita que prioridades aumentem quando a load aumenta.

Implementação do Escalonador

Em Unix tradicional:

- são mantidas 32 filas com as prioridades (VAX). `whichqs` contém um bitmask com um 1 para filas ocupadas.
- `swtch()` examina primeira fila, muda contexto, e quando retorna processo já está executando.
- Cada 100 ms (BSD) `roundrobin()` vai buscar outro processo com a mesma prioridade. Senão, o mesmo processo continua.
- `schedcpu()` é chamada de segundo a segundo para recomputar a prioridade.
- clock recomputa prioridade do processo corrente cada vez em 4.
- flag `runrun` é usada para indicar que processo de mais alta prioridade está à espera de ser executada, e é verificada antes de entrar em user-mode.

Problemas

Problemas do Escalonador

- Não escala bem: muitos processos faz com que recomputar prioridades seja pesado.
- Não se pode dar uma porção de CPU a um processo.
- Não há garantias de tempo de resposta para real-time apps.
- Aplicações não podem controlar as suas prioridades.
- Kernel nonpreemptive significa que processos de prioridade alta podem ter que esperar muito tempo antes de executar.

Objectivos do Escalonador de SVR4

Objectivos do desenho do escalonador em SVR4:

- Suportar mais aplicações, incluindo tempo-real.
- Separar a política de escalonamento dos mecanismos de escalonamento.
- Permitir às aplicações maior controle sobre prioridade e escalonamento.
- Definir uma interface bem estabelecida.
- Permitir a adição de novas políticas de uma forma modular.
- Limitar a latência de despacho para aplicações dependentes do tempo.

São fornecidas duas classes: time-sharing e tempo-real. Existem rotinas independentes de classe para fornecer mudança de contexto, manipulação da fila de processos e "preemption". Fornece uma interface para funções com herança e prioridades.

Processamento Independente de Classe

O nível independente de classe tem as seguintes características:

- Prioridades de 0 a 160, com filas separadas.
- Processo de maior prioridade corre sempre.
- Processos são colocados na fila por `setfrontdq()` e `setbackdq()` e removidas por `dispdeq()`.
- Para evitar latência de despacho (problema em Unix por o kernel ser nonpreemptive) define “preemption points”.
- Nesses pontos kernel testa `kprunrun` para ver se há processo tempo-real e tira o processo corrente.
- Exemplos são em parsing do pathname, na rotina `open()` antes de criar o ficheiro, e antes de libertar página.
- `runrun` existe: `preempt()` chama `CL_PREEMPT()` e depois `swtch()`.

Processamento Dependente de Classe

A componente dependente de classe é acedida como um vector de funções que implementam as componentes dependentes de classe.

Processos herdam classe do pai e podem ser mudados com `priocntl()`

- `proc` inclui ptrs. para id da classe, funções da classe e estruturas de dados privadas.
- `CL_TICK` é chamada do relógio.
- `CL_FORK` inicializa. `CL_FORKRET` inicializa `runrun` e permite ao filho correr primeiro.
- `CL_ENTERCLASS` e `CL_EXITCLASS` são chamadas numa classe.
- `CL_SLEEP` de `sleep()` e pode recomputar prioridade.
- `CL_WAKEUP` é chamada de `wakeprocs()` coloca processo na fila e pode colocar `runrun` ou `kprunrun`.

Prioridades em SVR4

Prioridades são divididas entre 0-59 para time-sharing, 60-99 para kernel e 100-159 para tempo-real.

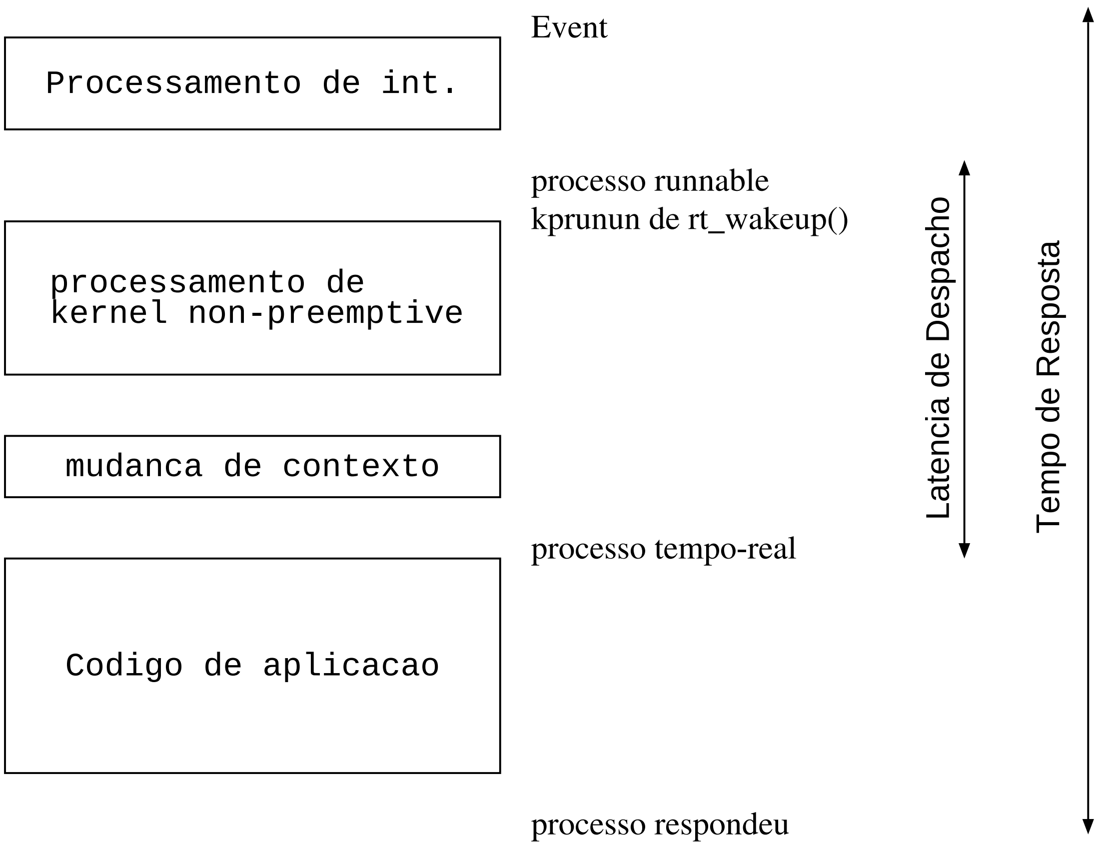
Time-Sharing em SVR4

Escalonamento é "round-robin" usando uma tabela de parâmetros fixa:

- Processos com menor prioridade têm maior time slice.
- Usa event-driven scheduling: prioridade é alterada na resposta a events.
- Dados dependentes de classe são `timeleft`, tempo para terminar o quantum, `cpupri`, a parte de sistema, `upri`, valor de `nice`, `umdpri`, $max(59, cpupri + upri)$ e `dispwait`.
- Em modo kernel prioriade é determinada pela condição de sleep, depois é restaurada de umdpri.
- Tabela de prioriades inclui o quantum, `cpupri` depois de quantum e de user mode, `lwait` para processos que usem mais do que `maxwait` segundos.

Processos de Tempo-Real em SVR4

- são especificados por `priocntl()`
- caracterizados por prioridade e quantum fixos.
- Processos exigem tempo de latência e tempo de resposta limitadas.



Análise de Escalonamento em SVR4

Novo algoritmo de escalonamento:

- Configurável por uma tabela.
- Não é preciso recomputar prioridades de todos os processos uma vez por segundo.
- Retuning pode ser necessário para manter equilibrio e evitar prejudicar processos interactivos com computação.
- É possível definir uma nova classe sem conhecer o código deo kernel.
- `priocntl()` é restrito ao superuser.
- real-time não é deadline-driven.
- Difícil encontrar prioridades certas: typing, batch, video e X. Necessário colocar video e X como tempo-real, mas prejudicava batch-jobs e sistema não respondia ao rato.

Escalonamento em Solaris

Solaris tem um escalonamento totalmente diferente:

- Kernel preemptive.
- Threads de interrupt permitem evitar `ipl`
- Suporte a multiprocessamento.
- Escalonamento Escondido
- Herança de prioridades
- Turnstiles.

Escalonamento para MP em Solaris

Suporte a Multiprocessadores inclui:

- Existe uma única fila de despacho.
- Threads podem ser restritos a um processador.
- Processadores podem enviar cross-processor interrupts.
- Cada processador mantém o `thread` executando, `dispthread`, o último thread executado, o `idle` thread, `runrun`, `kprunun` e `chosen_level`, prioridade do thread que vai tomar o processador.
- Se P1 tem um processo com maior prioridade que P2, coloca o seu `chosen_level` e envia-lhe um interrupt.

Escalonamento Escondido

O kernel pode fazer trabalho para threads:

- Kernel pode verificar pedidos em STREAMS, que são servidos com a prioridade do processo actual. Para evitar isso, STREAMs é feito em modo kernel, e abaixo de tempo real.
- Problema: pedidos de STREAMs feitos por processos de tempo-real?
- Call-outs têm o mesmo problema, por principio são executados em prioridade de interrupts.
- Solaris usa uma `callout` thread, que não inclui os `callouts` de real-time.

Inversão de Prioridades

Thread de baixa prioridade pode ser necessário para activar thread de alta prioridade:

- Herança de prioridade permite resolver o problema. Em Solaris thread tem *prioridade global*, dependendo da classe, e *prioridade herdada* que depende da interacção com objectos de sincronização.
- `pi_willto` é usada quando thread bloqueia para passear prioridade recursivamente para os donos de um objecto.
- Fácil para mutexes.
- Em geral impossível para semáforos e variáveis de sincronização.
- para readers-writers, Solaris usa *owner-of-record*, primeiro thread a ler o objecto.
- Herança de prioridades reduz tempo de espera, mas não evita que a cadeia de bloqueamento

cresça e muitos bloqueamentos em regiões críticas. Uma solução é usar um protocolo para garantir que processo com $>$ prioridade só bloqueia uma vez num recurso de threads menos prioritários.

Turnstiles

Muitos objectos de sincronização podem exigir muitos recursos ao sistema.

- Kernel tradicional usa “sleep channel”, um endereço, e usa esse endereço para procurar numa tabela de hash.
- Turnstiles são alocados dinâmicamente e só quando o objecto os necessita.
- Objectos de tamanho fixo que mantem os dados para sinc., como um ptr para a lista de threads bloqueados e para o dono.
- Threads bloqueados são colocados em ordem de prioridade e acordados por `signal()` ou `broadcast()`

Escalonamento em Mach

- Mach escalona threads independentemente de tasks.
- Prioridade por task + factor de uso por thread, decaindo a $5/8$ por segundo inactivo.
- Cálculos são feitos quando o thread acorda e com o relógio. Um thread interno recomputa prioridades de 2 em 2 segundos.
- Um thread pode correr mesmo quando prioridade nao é mais alta.
- *handoff scheduling*: thread pode passar controle a outro, útil para IPC.

Paralelismo em Mach

- Mach não usa interrupts entre processadores.
- Utilizadores podem criar *conjuntos de processadores*. Um servidor determina a alocação.
- É possível dedicar um conjunto de CPUs a uma task. Ou garantir que cada thread corre no seu CPU: *gang scheduling*
- Gang scheduling é útil para barreiras (pq nenhum thread se perde) e para aplicaçãoes fine-grained, pq podem atrasar num thread suspenso.
- Threads podem ser forçados a correr num CPU: útil para servidores sequenciais, ie de UNIX.
- Cada CPU tem uma fila local, e cada conjunto de processadores tem uma fila.
- CPUs favorecem a fila local. Útilpara servidor de Unix.

Escalonamento em Digital Unix

- Três classes: time-sharing, round-robin (prio. fixa) e FIFO (prio fixa, sem time-quantum).
- Escalonador escolhe o processo com > prioridade. Se processador preempted antes de terminar o quantum, colocado na frente da fila, senão atrás.
- Prioridades de threads são sobrepostas, dando flexibilidade. Acima de 19 processo precisa de superusr.

Escalonamento Paralelo em Digital Unix

- Escalonador é desenhado para optimizar mudanças de contexto e utilização de cache. Tb evitar problemas de luta por recursos quando vários processos são acordados da mesma fila global.
- Tenta recolocar threads no mesmo processador: *soft affinity*. Time-sharing threads usam filas locais a CPU. Sistema evita load imbalance.
- Processos com prio. fixa são escalonados de fila global, escalonador tenta reutilizar CPU.

Exemplos de Escalonamento

- Deadline Driven: vários tipos de deadline.
- Fair-share
- Multi-nivel: isocronico, tempo-real e time-sharing. Scheduler reserva recursos antes de os dar. processos de tempo-real têm um período máximo antes de devolver o CPU.
- Tarefas time-sharing são preemptible. Tempo real apenas em alguns pontos.

IPC em Unix

IPC permite transferência e partilha de dados, notificação de eventos, partilha de recursos e controle de processos (ex. debugging).

- Sinais: caros e apenas 31 em SVR4 ou BSD.
- Pipes: FIFO não-estruturado e unidirecional. Problemas: não permite broadcasts, não conhece limites de mensagens, não pode especificar o leitor.
  - Tradicionalmente: inode e entrada na tabela de ficheiros com pipe.
  - BSD usa sockets, SVR4 usa streams.
  - Named pipes: `mknod` é usado para criar o pipe, que depois é acessível a processos. Vantagens: persistência, acesso a qq processo. Desvantagens: têm que ser removidos, não são tão seguros, consome mais recursos, mais complicados de criar.
- Controle de Processos através de `ptrace(cmd, pid, addr, data)`.

- Permite a um processo ler ou escrever no espaço de um seu filho (incluindo área-u), mexer nos registos, interceptar sinais, criar *watchpoints* no espaço de endereçamento, resumir a execução de um filho parado, andar passo a passo, ou terminá-lo.
- cmd 0 é usado pelo filho para indicar que está controlada por `ptrace()`, alterando comportamento para sinais e para `fork()`
- Parente usa `wait()` para esperar eventos que mudam o estado da criança. Quando o evento chega usa `ptrace()` para controlar.
- Limitações: só pode controlar filhos imediatos; não permite apanhar processos em andamento, extremamente ineficiente, problemas com programs `setuid()`.
- `/proc` é a alternativa usada em sistemas modernos.

IPC em SYSV

SYSV suporta *semáforos*, *filas de mensagens*, e *memória partilhada*

Cada recurso tem os seguintes atributos:

- Chave: inteiro que identifica a instância do recurso.
- Criador: UID e GID do processo que creou
- Dono: pode ser <> do anterior.
- Permissões.

`get` cria o recurso, `cft` controla com `STAT`, `SET`, `RMID`.

Cada recurso tem uma tabela de tamanho fixa.

Semáforos em SYSV

- `semid = semget(key, count, flag)` arrayt de count semáforos.
- `status = semop(semid, sops, nsops)`, onde sops aponta para um array de operações. Operação pode ser incrementar (> 0), esperar até semáforo estar a 0 ($= 0$), ou esperar que o valor seja maior ou igual ao valor absoluto (< 0) e depois subtrair esse valor.
- Todas as operações avançam ou bloqueiam. Nenhuma outra operação pode executar em paralelo.
- `IPC_NOWAIT` evita bloqueio.
- Kernel mantém `UNDO LIST` para o caso do processo sair.

IPC em Mach

Troca de Mensagens é o principal mecanismo de comunicação:

- Mensagens podem variar entre alguns bytes e um espaço de endereçamento.
- Comunicação deve ser segura.
- Comunicação ligada a gestão de memória.
- Comunicação entre user tasks, e com o kernel.
- Suportar o modelo cliente-servidor
- Interface pode ser generalizada para ambiente distribuido.

Portas em Mach

Uma porta tem direitos de send e de receive (apenas o dono): comunicação muitos-para-um.

- Cada porta tem um contador de referências para porta. Cada direito ou capacidade, é um nome para a porta. Nomes são inteiros e *locais* a cada task.
- Cada objecto é representado por uma porta. Acesso a essa porta permite ao dono fazer operações no objecto. O kernel tem os direitos de recepção para essas portas.
- Cada porta tem uma fila de mensagens finita. Emissores bloqueiam quando a fila enche.
- Cada task tem uma porta *task_self* para ela própria, e direitos de recepção para *task_notify*. Pode enviar para *bootstrap* que fornece acesso ao name server. Cada thread tem direitos de envio para `self` e recepção para `reply`. Existe uma `exception` para todos os threads e tasks.

Mensagens em Mach

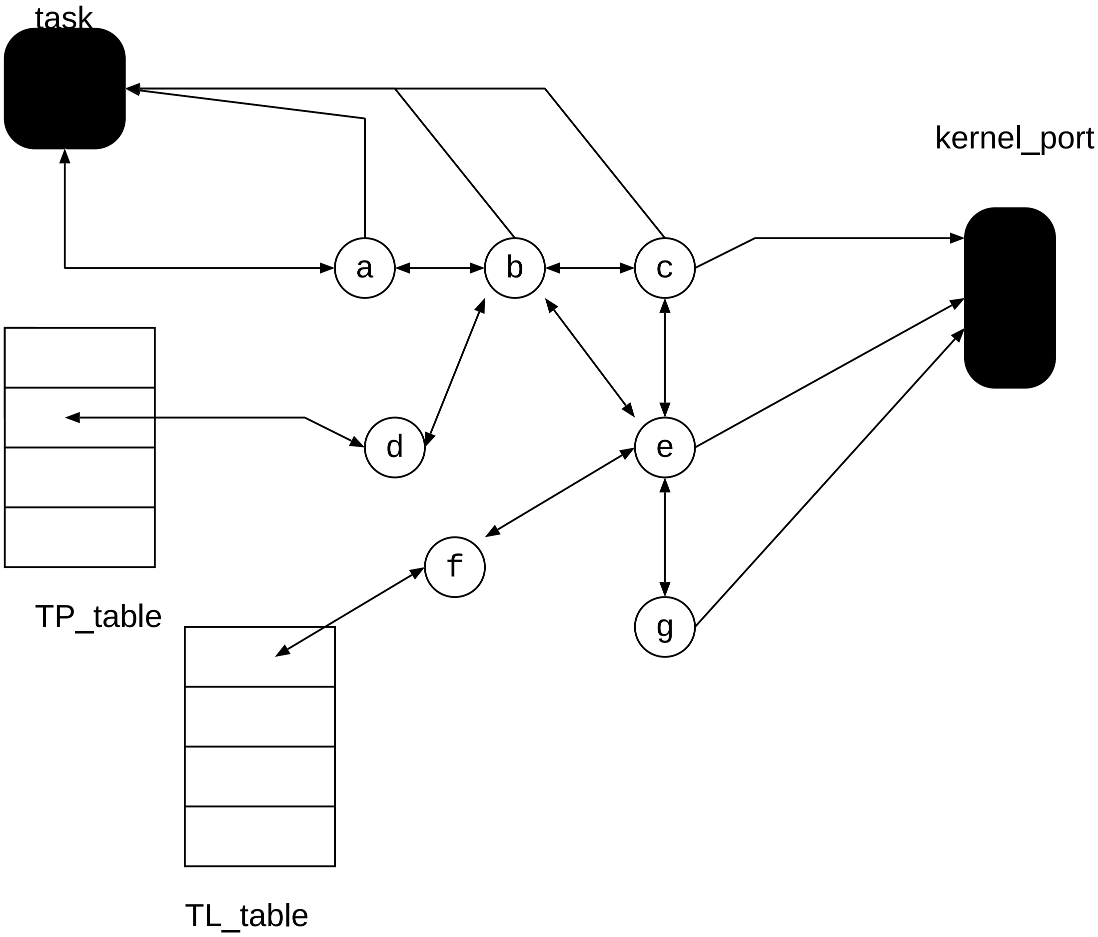
Mensagens podem ser locais ou por rede (através de `netmsgserver`):

- Mensagem contém dados, memória out-of-line, e direitos de envio ou recepção para portas.
- Cabeçalho contém o tipo, tamanho, porta destino, porta de resposta, ID.
- Components contêm o tipo, tamanho, número de items, e flags.
- `msg_send()` envia sem esperar, `msg_rcv` espera por mensagens, `msg_rpc` envia e espera por uma resposta que pode vir no próprio buffer. Operações têm `TIMEOUT`.

Implementação de Portas em Mach

Portas são filas protegidas de mensagens:

- Cada porta é representada por uma estrutura de dados no kernel. Cada task pode ter no máximo um nome para uma porta, que é um inteiro (semelhante a Unix fd).
- A estrutura de dados para uma porta contém contador de referências para a porta, ptr para a task que tem direitos de recepcão, nome local no receptor, ptr para porta backup, dbl lista de mensagens, fila de emissores bloqueados, fila de threads receptores bloqueados, lista de todas as traduções, ptr para um "port set", núm. de mensagens na fila, núm max. permitido.



- Traduções podem ser acedidas por send $< task, local\_name >$ ou por rcv como $< task, port >$. Tasks tb tem que ser capazes de encontrar todos os direitos para a porta quando ela é dealocada. Finalmente, direitos têm que ser limpos quando a porta é destruída.

Passagem de Mensagens em Mach

- Emissor cria mensagem; chama `msg_send()`; kernel copia mensagem e (i) se thread está à espera é acordado e recebe, (ii) se lista cheia emissor bloqueia, senão mensagem colocada na fila; kernel regressa de `msg_send()`.
- receptor chama `msg_rcv()`, kernel chama `msg_dequeue()`, kernel copia para receptor.
- Se emissor espera resposta, envia direitos para porta usando campo *reply port* na mensagem.
- Servidor de nomes também pode dar direitos de acesso a uma porta de um servidor. O servidor de nomes tem direto de acesso à maioria dos servidores no sistema.
- Quando nova porta chega kernel deve fazer tradução, verificando primeiro se porta já existe, senão aloca um novo índice e cria um novo nome.
- Para transferências grandes Mach usa

copy-on-write. Se emissor usar deallocate, `copyin()` limita a passar as páginas para receptor.

- Notificações: mensagens enviadas para informar uma task de eventos.
- Mensagens podem ser enviadas no caminho lento (colocar na fila) ou caminho rápido (handoff scheduling).

Operações sobre Portas em Mach

- Destruição de portas: mensagens são removidas e notificações são enviadas. Se mensagem contém direitos sobre a própria porta dá confusão.
- Portas backup: usadas quando a porta original é destruída.
- Conjuntos de Portas: um receptor recebe todas as mensagens para o conjunto. Permite controle de várias portas por uma única task.
- Interpolação de portas: permite substituir uma capacidade para uma única porta com uma porta diferente. Permite a um debugger controlar acesso a um processo.

Passagem de Mensagem em Rede

`netmsgserver` permite extensão para rede:

- Usa proxy ports para “enganar” clientes, e comunica com outros netmsgserver para distribuir o sistema.
- Possível porque cliente tem apenas acesso a nome local para a porta, e porque emissores são anónimos: o emissor pode enviar apenas o direito de acesso a uma porta de resposta.

Mach 3.0

- Dificil dealocar send rights, pq não se sabe que threads estão a usar o direito: send-once rights.
- Apenas envia notificações a processos que as pediram.
- Kernel mantém um contador de referência a direitos por task. Quando o contador vai a 0, pode dealocar.

Sincronização em Multiprocessadores

Multiprocessadores oferecem várias vantagens:

- Expansibilidade: adicionar mais CPUs.
- Aumentar CPU sem aumentar outros recursos.
- Fiabilidade: "fault tolerant systems".

Paralelização de Unix exige muitas alterações:

- Modelo de sincronização não funciona.
- Colocação e granularidade de locks.
- Escalonamento

Sincronização Tradicional

Características:

- Reentrante: vários processos podem estar no kernel ao mesmo tempo.
- Nonpreemptive: threads correm até sair do kernel, evita locking.
- Interrupts controlados com `ipl`
- recursos partilhados são controlados por flags `locked` e `wanted`.
- Recursos são mapeados numa *sleep queue*: eventos diferentes bloqueiam nos recursos, e vários recursos mapeiam na mesma entrada.

Suporte a Multiprocessamento

Baixo Nível

- Atomic test-and-set: atómicamente retorna o valor antigo do bit e coloca o novo valor a um.
- Extensão: fazer isso com uma palavra. `LDSTUB` e `SWAP` no SPARC e MC88100.
- `LL` e `SC` no MIPS e ALPHA

Nível do Kernel:

- Master-Slave: mestre pode ser único a realizar I/0 e a receber interrupts.
- Assimétricos funcionalmente: processadores especializados.
- Simétricos.

Problemas com Sincronização em MPs

Mecanismo de Unix não funciona:

- Vários threads podem aceder a `locked` simultaneamente.
- Bloqueamento de Interrupts não funciona.
- Wakeup perdido: enquanto um processo pode estar a adormecer, outro processo pode recuperar o recurso. O primeiro processo pode bloquear para sempre.
- Thundering Herd: vários processos bloqueados no mesmo recurso podem acordar ao mesmo tempo, e ser escalonados para diferentes CPUs, competindo pelo mesmo recurso.
- Starvation: um processo pode perder o recurso.

Semáforos

Usados nas primeiras implementações:

- P (down) e V (up). Pode haver versão não bloqueante de P.
- Exclusão mútua: inicialmente a 1. Faz P, usa, V.
- Espera de eventos: inicializa a 0.
- Recursos numeráveis: inicialmente o número de recursos disponíveis.
- Bloquear em semáforos pode ser lento porque exige manipulação de filas. Alem disso, semáforos não dão garantia sobre o que estão a proteger.
- Comboios: problema típico de semáforos.

  Processos podem bloquear desnecessáriamente ficando à espera que processos acordados num semáforo sejam escalonados.

Spin-Locks

Mutexes que fazem busy-wait. Para operações rápidas:

```
void spin_lock(spinlock_t *s) {
  while (test_and_set(s) != 0) ;
}
```

Evitar tráfego desnecessário:

```
void spin_lock(spinlock_t *s) {
  while (test_and_set(s) != 0)
    while (*s != 0) ;
}
```

- Bloqueiam CPU: usados por tempo curto.
- Uniprocessadores podem bloquear se spin-lock tem disable de interrupts.
- Usados para implementar semáforos.

Variáveis de Condição

Associadas a um predicado baseado em dados partilhados.

- `wait`: espera pelo recurso
- `signal` e `broadcast`: acorda um ou todos os processos.
- Semelhante a um sleep channel. Para evitar problemas com a condição é necessário um mutex.
- Obriga a um processo manter vários locks: não há problema se os locks forem ordenados.
- Event: variável `done`, spin-lock protegendo a variável, e uma variável de condição.
- Blocking locks podem ser implementados com variáveis de condição.

Locks de Leitura Escrita

- LockExclusive e LockShared
- UnlockExclusive e UnlockShared
- Outras: tryLock, upgrade e downgrade.
- O que fazer quando se liberta um lock?
- Ultimo leitor deve acordar um escritor.
- Um escritor pode acordar leitores (um ou todos) ou outro escritor.
- Muitos leitores podem bloquear escritor: bloquear se há escritor.

Considerações

- Contador de Referências: necessários para quando se partilham objectos.
- Prevenção de Deadlock: locking hierárquico e estocástico.
- Exemplo: buffer e lista.
- Locks Recursivos: processo que já tem um lock pode voltar apedi-lo (UFS): ter um campo dono.
- Bloquear ou rodar? Depende da duração e de quem tem o recurso. Hints. Solaris tem locks adaptativos.
- Granularidade e Duração.

Implementações

- Digital Unix Locks Simples e Complexos
- Também suporta sleep e wakeup
- SVR4.2 MP suporta mutexes (com IPL), RW locks, sleep locks e variáveis de sincronização.
- Solaris usa locks adaptativos, turnstiles, e fornece semaforos, RW locks e variáveis de condição.

Ficheiros em Unix

- Abstrações: ficheiros (arquivos), directórios, descriptor de ficheiro e sistema de ficheiros.
- Ficheiro contem dados. Em Unix dados são uma sequência ordenada de bytes. É responsabilidade da aplicação estruturar os dados.
- Ficheiros são organizados hierárquicamente com *directórios* sendo os nós da árvore.
- Nome de ficheiros podem conter qualquer caracter excepto "/" e o caracter nulo.
- Unix suporta *cwd*, caminhos absolutos e caminhos relativos.
- Entrada numx directorio é um *hard link*. Todos os hard links são equivalentes em Unix.
- Unix moderno fornece `opendir()`, `readdir()`, `rewinddir()` e `closedir()` que operam com `dirent`

Atributos de Ficheiros

- Tipo do Ficheiro: normal, directório ou especial.
- Número de hard links.
- Tamanho.
- Device + Nó-i
- UID e GID do dono
- Timestamps: último acesso, última modificação, uúlt. mod. dos atributo.
- Permissões: leitura, escrita, acesso. Alt: ACLs.
- Flags: suid, sgid, sticky. sgid é usada para locking em aruivos não executáveis. SVR4 usa sgid para dizer de onde se hera o GID (criador (SYSV) ou dir (BSD). Sticky é usada para restringir remoções de ficheiros.
- syscalls: `link`, `unlink`, `utimes`, `chown`, `chmod`.

Descriptores de ficheiros

- `fd = open(path, oflag, mode)`, `creat`: `O_WRONLY`, `O_CREAT` e `O_TRUNC`.
- Processo tem uma default file creation mask alterável por `umask()`
- Em `open()`, kernel cria um *open file object* e aloca um *file descriptor*. O mesmo ficheiro pode ser aberto várias vezes.
- FD existe por processo e permite evitar procurar o objecto. Representa uma sessão. Contexto inclui modo e offset. Vários objectos para o mesmo ficheiro podem existir.
- `dup()` e `dup2()` duplicam fd.
- Processos podem passar fd para outros processos (referência a objecto). SVR4 usa streams, BSD usa sockets com `sendmsg()`. Usado por *connection servers*.

I/O

Acesso pode ser sequencial ou “random”.

- `read()` e `write()`. `read()` lê no máximo `count` caracteres e copia-os para `buf`.
- operações são atómicas entre elas.
- Solaris fornece `pread()` e `pwrite()`
- `readv()` e `writev()` implementam *scatter-gather I/O* por eficiência.
- Locking pode ser mandatório ou “advisory”.
- BSD inclui `flock()`, com locks partilhados e exclusivos, mas só advisory. SVR2 suporta advisory para files e records. SVR3 inclui locking mandatório, via `chmod()`. SVR4 suporta BSD através de `fcntl()` e de `lockf()`.

Sistemas de Ficheiros

- Sistemas de ficheiros têm uma hierarquia. `mount()` coloca um sistema de ficheiros sobre outro e esconde o que estava antes.
- Disco lógico abstrai armazenamento: disco, partição. Permite também suportar espelhos de discos, striping, e RAID.

Ficheiros Especiais

`fstat()` permite verificar o tipo de um ficheiro:

- Links simbólicos: evitam alguns problemas de hard links como links para directórios e problemas de proteção.
- Consistem de um "pathname" que pode estar no próprio nó-i ou num bloco especial.
- pipes e FIFOs.
- devices que podem ser de caracteres ou de blocos.

Múltiplos Sistemas de Ficheiros

Torna-se necessário suportar vários sistemas de ficheiros (s5fs, ufs, fat, NFS, etc).

- I/O mostra que kernel pode suportar diferentes dispositivos com o mesmo framework. A ideia é que o sistema usa as operações gerais enquanto pode. Quando é necessário fazer algo de específico existe uma array de operações (open, close, read, write) em que cada dispositivo fornece a sua rotina (ou usa uma geral).
- A mesma ideia pode ser usada para suportar múltiplos sistemas de arquivos.
- A ideia é uma aplicação dos conceitos de programação por objectos.

Vnode/Vfs

A Sun desenvolveu esta interface:

- 2 classes: vnode e vfs.
- Cada vnode tem dados independentes do FS, operações virtuais que definem a interface, um conjunto de utilitários usados pelo resto do kernel, e dados e operações que dependem do FS.
- Estruturas `v_data` e `v_ops` implementam as componentes dependentes do FS.
- vfs são semelhantes.
- em C implementados como `struct`, macros e ptrs. para outros `structs`.

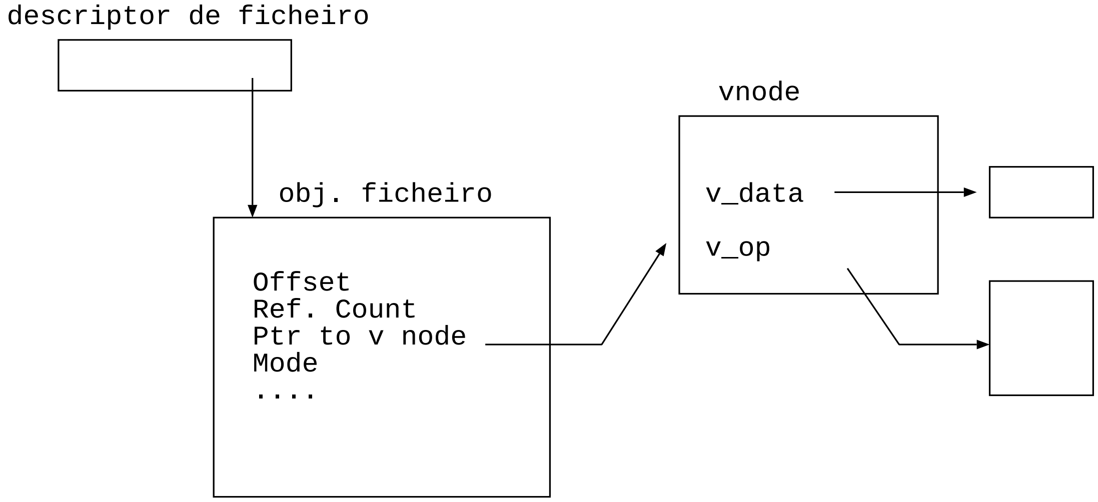
Implementação de Nós-V: objectivos

Ideia é poder usar a interface em sistemas muitos diferentes. Isso obriga a que:

- Cada operação deve poder ser realizada em função de um processo que adormece se a função bloqueia.
- Operações que necessitem de serialização devem poder fazer locking das estruturas de dados.
- A interface deve ser "stateless", evitando variáveis globais e campos na área u para passar info. entre operações.
- a interface deve ser reentrante, evitando variáveis globais, especialmente para erros.
- Implementações devem poder usar recursos globais como a cache de buffers.
- Interface deve ser usável por um servidor.
- Evitar tabelas de tamanhos fixos.

Relação entre nós-v e ficheiros abertos

Processo pode aceder um nó-v ou via lookup ou através de um fd.



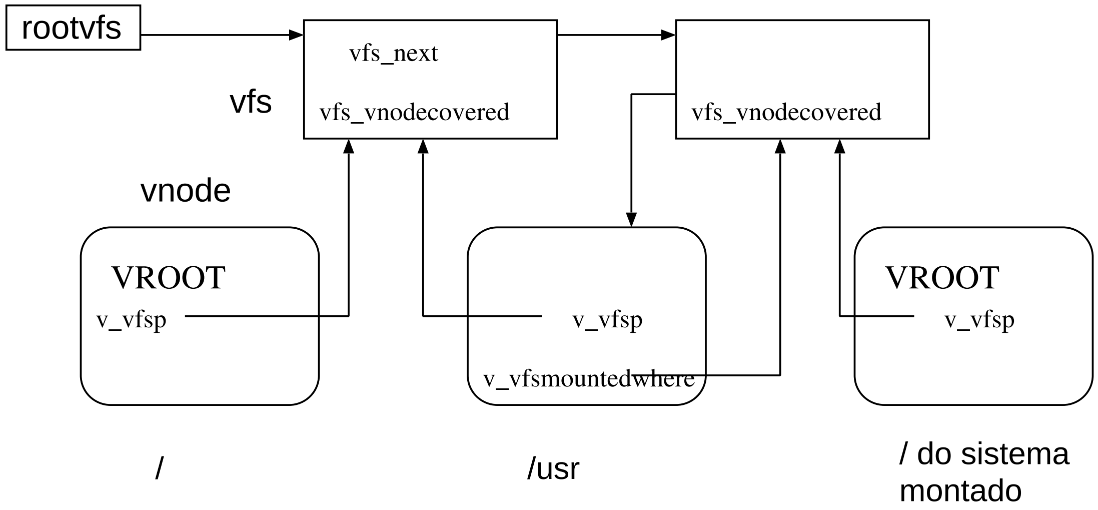
- Antigamente fd era alocado num array estático
- Hoje em dia é alocado dinâmicamente, por exemplo como uma lista ligada de blocos de 32 entradas.
- Outra hipótese é aumentar com `kmem_realloc()`

Implementação do Nó-V

Cada nó-v tem as seguintes estruturas:

- flags, indicando se é raiz de um FS, etc.
- número de referências (ficheiros abertos, cdw, mount points, lookup). Importante para arquivos temporários.
- ponto de montagem.
- operações.
- file system.
- Stream
- Páginas residentes.
- Tipo.
- device ID.
- dados privados

Objecto V-vfs



Campos:

- vfs_next: VFS seguinte na lista.
- vfs_op: vector de operações.
- vfs_vnodecovered: nó onde vfs está montado.
- vfs_dev e vfs_vfstype: ID do dispositivo e index para tipo de file system.
- vfs_data: dados privados ao FS.

Campos Dependentes da Implementação

- Dados privados: opacos e alocados juntamente com parte independente.
- Operações da interface no vnode: `open()`, `close()`, `read()`, `write()`, `ioctl()`, `getattr()`, `setattr()`, `access()`, `lookup()`, `create()`, `remove()`, `link()`, `rename()`, `mkdir()`, `rmdir()`, `symlink()`, `readlink()`, `inactive()`, `rwlock()`, `rwlock()`, `rwunlock()`, `realvp()`, `getpage()`, `putpage()`, `map()`, `poll()`.
- Operações da interface no vfs: `mount()`, `unmount()`, `root()`, `statvfs()`, `sync()`.

Montagem de FS

- Em SVR4 existe uma tabela global com as características de cada FS.
- `mount` primeiro obtem o vnode do ponto de montagem com `lookupn()`, obtendo uma entrada e um lock.
- Chama `vfs_init()`, específico ao FS.
- Inclui estrutura na lista ligada comandada por rootvfs.
- Instala `vfs_op` e `vfs_vnodecovered`
- chama `VFS_MOUNT` que verifica permissões, aloca e inicializa estrutura de dados privados, coloca ptr em `vfs_data`, e inicializa o vnode da root do FS.
- Sistemas locais usam superbloco, sistemas distribuidos chamam o servidor.

Travessia do nome

`lookupn()` recebe um nome e retorna um ptr para vnode:

1. Verifica se o nó-v inicial é directório (se não for final).
2. Se componente é ".." e cwd raíz, apanhe o componente seguinte.
3. se componente é ".." e cwd VROOT, acesse mountpoint.
4. se componente não fôr encontrado e fôr último, retorne sucesso, passando ptr para o pai e mantendo hold. Senão ENOENT.
5. se componente novo é um ponto de montagem encontre o vfs correspondente e chame vfs_root para encontrar o vnode raíz.
6. Se v_type == VLNK, traduza com (VOP_SYMLINK), junte a tradução e reinicialize (se caminho absoluto, comece da raíz).

`lookupn()` pode desactivar avaliação de links simbólicos. Existe tb limite máximo para evitar ciclos.

7. Liberta directório (segurado ou por `VOP_LOOKUP` ou por inicialização.
8. Se encontrou, mantenha o hold e regresse um ptr para o vnode. Se caminho esgotado e não encontrou termine. Senão volte ao ciclo.

Cache de Acesso a Directórios

- Cache LRU contendo directório, ficheiro no directório, e ptr para vnode do ficheiro.
- Organizado por dir+entrada.
- `VOP_LOOKUP` procura lá primeiro.
- Para garantir validade contra remoções de ficheiros, em SVR4 cache tem uma referencia paa o nó-v, o que impede remoções e também impede uso exclusivo por outras rotinas.
- Em 4.3BSD cada nó-i tem uma capacidade, que é incrementado sempre que o nó-i é entregue a um ficheiro novo. A cache também tem uma capacidade, que é comparada com a do nó-i em acesso.

VOP_LOOKUP

`erro = VOP_LOOKUP(vp, compname, &tvp,` . . ., onde vp é ptr para nó-v do pai, compname é o componente, e tvp o resultado.

- Geralmente, primeiro procura na cache, se encontrar retorna nó-v e incrementa referências.
- Se não encontrar itera no directório pai até encontrar o nome (se local), ou envia pedido a servidor (remoto).
- Verifica se nó-v correspondente está em memória (tabela de hash).
- Se não estiver aloca o nó-v.

OPEN

A rotina `open()` funciona assim:

1. aloca um fd.
2. aloca um objecto ficheiro.
3. chama `lookupn()` para encontrar o nó-v.
4. Chama `VOP_ACCESS` para verificar permissões.
5. Verifica se operação é ilegal (abrir um directório ou executável activo para escrita).
6. Se `O_CREAT` e ficheiro não existe, chama `VOP_CREATE` no directório pai, senão erro.
7. Chama `VOP_OPEN`, que geralmente não faz nada.
8. Se `O_TRUNC`, chama `VOP_SETATTR` para colocar tamanho a 0.
9. Inicializa o objecto ficheiro.
10. Retorna o fd.

Comentários

- kernel locka o nó-v antes de fazer leitura ou escrita.
- Atributos são implementados por estrutura `vattr`, baseada no s5fs.
- Em SVR4 e Solaris, credenciais são passadas por referência a um objecto.
- Vantagens: portátil, geral.
- Desvantagens: `lookupn()` chama `VOP_LOOKUP` para cada componente, devido a NFS.
- operação não faz lock do directório pai: possível a outro processo criar o arquivo que queremos criar, causando overheads em `VOP_CREATE`.
- BSD resolve o problema passando um parâmetro extra com a razão do `lookupn()` e fazendo locking se create. Problema é que serializa operações no directório.

- OSF usa timestamps para verificar se o directório não foi alterado.
- Continua a haver dependências com a parte de gestão de memória. Interface varia com diferentes UNIX.

Partições s5fs

B	S	Lista de No-I	Blocos com Dados

- Directório: lista linear de registos com 16 bytes. Primeiros 2 contêm nó-i, e os outros 14 o nome do ficheiro. As primeiras entradas são sempre "."e "..". Entrada a 0 indica que não existe.
- Cada arquivo tem um nó-i, com o as permissões, número de hard links, UID e GID do dono, tamanho, array de blocos, número de geração, e alterações.
- Ptrs são directos, indirectos, duplamento indirectos, e triplamento indirectos, dando um total de 16 milhões de blocos (tamanho de 1KB).
- Podem existir buracos, o que causa problema para **tar** e **cpio**.

O Superbloco

Inclui:

- Núm. de blocos no sistema de ficheiros.
- Tamanho da lista de nós-i. Também suporta sleep e wakeup
- Núm. de blocos e nós-i livres, variáveis de sincronização.
- Lista parcial de nós-i livres: se acabar kernel procura nos nó-i.
- Lista de blocos livres que pode cobrir vários blocos. Kernel recupera blocos quando núm. de blocos livres diminui.

Organização no Kernel

- Nó-i em memória inclui: vnode, devID, núm. do nó-i, flags, ptr para uma lista de nós-i livres e para uma fila de hash, núm do últ. bloco a ser lido.
- `iget()` procura um nó-i pelo seu número e inicializa-o se não existir.
- Blocos poderm ser acedidos via buffer cache ou via `getpage()`
- Quando o núm de ref. vai a 0, o kernel pode libertar o nó-i. Melhor só libertar se não estiver com nenhum buffer.

Análise de s5fs

- A maior vantagem é simplicidade.
- Desempenho pode ser mau porque nós i estão longe dos dados correspondentes.
- Alocação de blocos torna-se rápidamente aleatória.
- Tamanho de blocos demasiado rígido.
- Limites na funcionalidade: tamanho máximo do ficheiro.

Introdução a FFS

- Discos divididos em “grupos de cilndros”. Superbloco é dividido em 2 partes: informação sobre o FS e sobre o grupo.
- Superbloco é copiado para cada grupo.
- Blocos são divididos como blocos (8k) e fragmentos (1K ou 512B). Apenas o último bloco pode ter fragmentos, o que pode obrigar a cópia.
- Politica de alocação favorece grupos de cilindros.
- Nomes longos e links simbólicos.

Sistemas de Ficheiros Distribuídos

Propriedades:

- Transparência de rede: mesmas operações que num sistema local.
- Transparência de localização: nome não deve revelar localização.
- Independência de localização: nome não deve mudar quando colocação física mudar.
- Mobilidade do utilizador: utilizadores devem poder aceder a partir de qualquer nó.
- Tolerância a Falhas: continuar a funcionar perante a falha de um componente.
- Escalabilidade: escalar quando a carga aumenta, e ser expandido incrementalmente.
- Mobilidade de Ficheiros: mudar de localização física.

Considerações de Desenho

- Espaço de nomes: uniforme ou adaptável por cada cliente.
- Servidor com ou sem estados. Ex: offset no servidor para leitura ou cada cliente envia. Não ter estados simplifica consistência e recuperação de falhas.
- Métodos de Acesso Remoto: sempre chamado pelo cliente ou com uma participação mais activa do servidor.

Desenho de NFS

Originalmente desenvolvido pela Sun, suporta sistemas Unix e não Unix.

- Baseado num modelo cliente servidor.
- Comunicação via RPC síncrona.
- Cada servidor exporta um ou mais sistemas, para leitura ou leitura-escrita.
- Montagem pode ser *hard*, tenta até conseguir, ou *soft*, desiste após um bocado, ou *spongy*, hard no princípio e depois soft.
- Cliente pode não ser privilegiado, e pode montar o mesmo sistema em vários locais.
- Servidor apenas pode exportar FS locais e não pode atravessar pontos de montagem.

Objectivos de NFS

Os objectivos da Sun eram:

1. NFS não ser restrito a Unix (tanto servidores como clientes).
2. Protocolo não devia depender de hardware.
3. Mecanismo de recuperação simples.
4. Acesso transparente a ficheiros remotos.
5. Manter semânticas de Unix para clientes Unix.
6. Desempenho comparável a disco local.
7. Implementação independente de transporte (UDP, TCP).

Componentes de NFS

Componentes principais:

- NFSv2: protocolo definindo operações e seus argumentos. Pedidos incluem `NULL`, `GETATTR`, `SETATTR`, `LOOKUP`, `READLINK`, `READ`, `WRITE`, `CREATE`, `REMOVE`, `RENAME`, `LINK`, `SYMLINK`, `MKDIR`, `RMDIR`, `READDIR`, e `STATFS`. Argumentos podem ser uma fhandle, dirfh, offset.
- RPC: define o formato das interacções entre o cliente e o servidor. Pedido NFS é um pacote RPC.
- XDR: Extended Data representation usada por RPC para codificar dados.
- Código de implementação do servidor NFS.
- Código de implementação do cliente NFS.
- Protocolo de montagem: `NULL`, `MNT`, `DUMP`, `UNMT`, `UNMTALL`, `EXPORT`.
- Processos daemon: *nfsd* e *mountd* no servidor,

e *biod* suporta I/O assíncrono no cliente.

- O Network Lock Manager e o Network Status Monitor permitem locking de ficheiros.

Statelessness

Todos os pedidos são independentes:

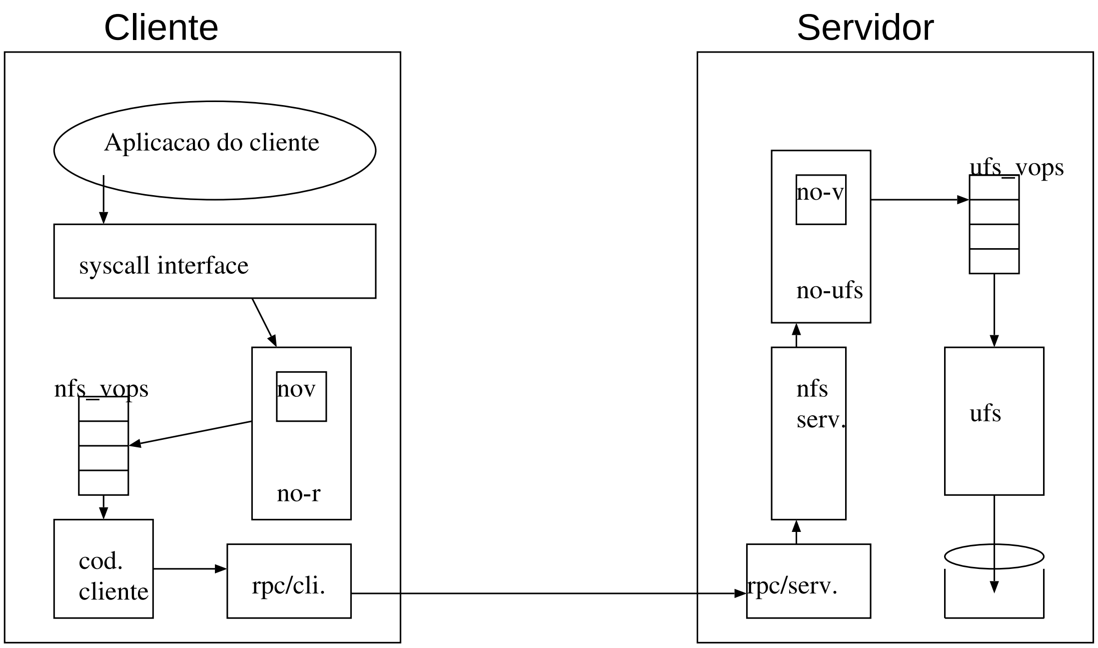
- não há pedidos para abrir e fechar ficheiros. Registos de pedidos só para estatísticas ou para caching.
- Se um servidor faz reboot, o cliente continua a enviar os pedidos até que o cliente faz reboot, e nessa altura o servidor pode devolver a informação.
- O servidor deve colocar em armazenamento estável todas as alterações antes de responder (dados de ficheiros, dir., inode, etc.)

O Conjunto de Protocolos

Os protocolos mais importantes são:

1. XDR usa uma representação tipo Sun com inteiros, objectos opacos, strings, vectores e estruturas.
2. RPC da Sun é um protocolo síncrono de comunicação entre cliente e servidor. É seguro apesar de habitualmente ser implementado sobre UDP. Campos incluem xid, direcção, rpc_vers, programa e sua versão, informação e autenticação.
3. Autorização pode ser `NULL`, `UNIX`, `SHORT`, usada depois do primeiro pedido Unix, `DES`, e `KERB`.

Implementação de NFS



- File Handles: evitam ter sempre que abrir o ficheiro. Incluem o fsid, nó-i e o número de geração.
- Montagem comunica com mountd. Servidor NFS mantém uma lista de directórios exportados, para verificar rápidamente quais as permissões.
- lookup.

Compatibilidade com Unix

- Em Unix, util. pode sempre escrever num ficheiro aberto, mesmo que o dono tenha alterado as permissões. NFS permite ao dono escrever e ler sempre de um ficheiro, é o cliente que segue as operações.
- Ficheiros removidos continuam acessíveis em Unix. Em NFS cliente muda a operararação para um rename, e no fim um remove. Problema: cliente pode crashar.
- Reads e writes podem involver várias operações e não são portanto atómicos entre clientes: usar NLM.

Performance de NFS

- Operações que alteram o servidor são extremamente lentas porque esperam que os writes sejam colocados no armazenamento. Além disso, obter atributos obriga a um RPC por ficheiro: problema com `ls -l`. Retransmissões podem piorar situação em redes sobrecarregadas.
- Clientes fazem caching de dados. Perigoso: atributos são guardados no máximo de 1 min, e blocos de dados são válidos se a data de alteração não tiver sido mexida.
- Cliente usa writes assíncronos para blocos completos, e writes atrasados para blocos parciais.
- Servidor pode usar NVRAM em vez de disco. Driver pode optimizar transferências de NVRAM para disco.
- Write-Gathering: servidor espera em vez de

processar todos os writes de uma vez. Útil se vários biods no cliente.

Retransmissões

- Pacotes perdidos podem acontecer com UDP.
- Podem afectar operações *idempotentes*, que podem ser executadas várias vezes, ou *não idempotentes*, como `CREATE` ou `REMOVE`.
- Solução: manter uma cache de pedidos que podiam dar problemas. Se pedido falha, verifica se é uma retransmissão e está na cache.
- Solução é incompleta: Digital faz cache de todos os pedidos.

NFS: Extensões

- Existem servidores dedicados, alguns dos quais paralelos e com FS muito diferente de UFS.
- Problemas de segurança: UID e GID devem ser os mesmos em todo o grupo.
- Problemas são limitados com UID remapping.
- Melhoramentos em NFSv3, como writes assíncronos, evita pedidos desnecessários, `READDIRPLUS`, e $>$ tamanho máximo.

Problemas de Memória Virtual

Vantagens:

- Executar programas maiores que o tamanho da memória.
- Executar programas carregados parcialmente.
- Permitir mais do que um programa ao mesmo tempo.
- Permitir código independente de posição.
- Libertar programadores da aloca‘cão de memória.
- Permitir partilha

VM em Unix

- Originalmente segmentos.
- Swapping usado para libertar espaço.
- Primeira implementação com paginação: 3BSD e VAX 11.
- Nova versão em SunOS 4 influenciou o desenvolvimento de paginação para SVR4 e Solaris.
- Paginador para Mach foi eventualmente adaptado para BSD4.4.

Paginação em Solaris

Ideias:

- Mapeamento de Ficheiros (privados e partilhados).
- Unificação de Acesso a ficheiros e de memória virtual: evitar chamadas excessivas a SO.
- Permitir swapping dinâmico em ficheiros.
- Partilha de Memória entre processos: read-only e read-write.
- Estrutura orientada para objectos.

Desenho de VM em Solaris

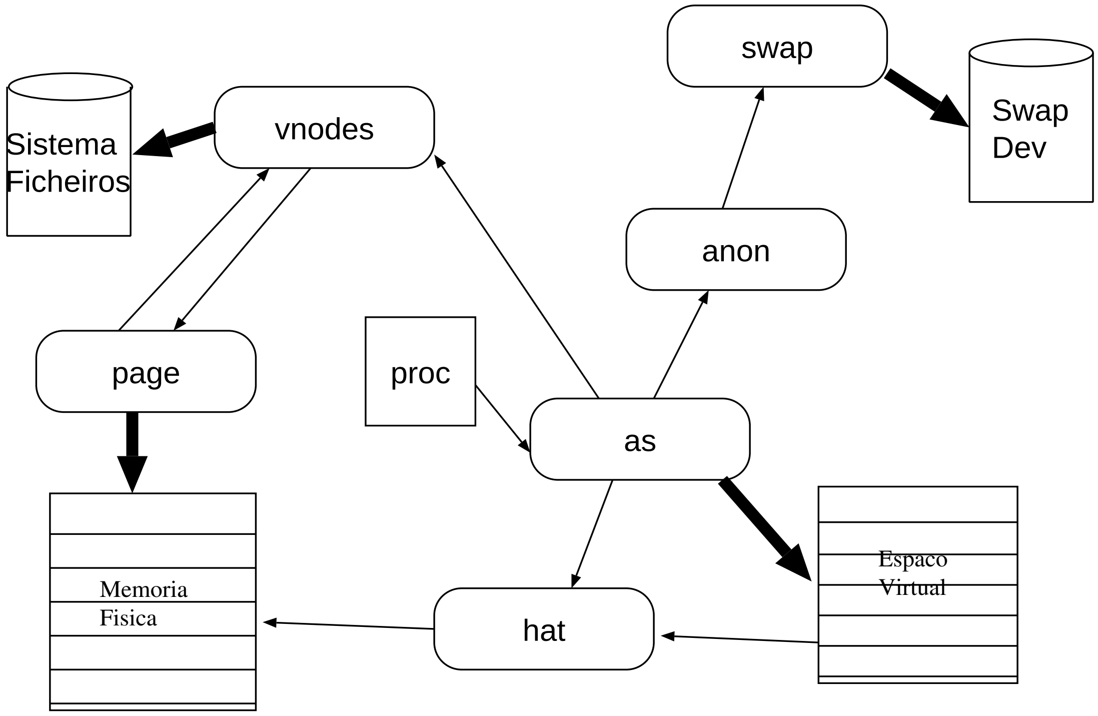
Desenho orientado para objectos:

1. Cada tipo de objecto de memória é uma sub-classe de uma classe base. Cada mapeamento é uma instância de uma classe base.
2. O espaço de endereçamento é constituido pro um conjunto de mapeamentos para diferentes objectos. O objecto mapeado não sabe do mapeamento.
3. Objectos de memória podem ser associados a um nó-v (muitos para um). Objectos não associados a nós-v são representados pelo *objecto anónimo*.
4. Memória é organizada como conjunto de páginas, que é a unidade alocaçao, proteção, tradução de endereços e mapeamento.
5. Páginas físicas servem como cache para objectos pageados.

6. Arquitectura de VM é independente de Unix.
7. Kernel usa copy-on-write quando possível.

Abstrações Fundamentais de VM

O sistema usa `page`, página, `as`, espaço de endereçamento, `seg`, segmento, `hat`, tradução de endereços em hardware, e `anon`, páginas anónimas.



- Memória física: cada página pertence a um nó-v. Cada pág. inclui informação sobre o nó-v, o seu offset, ptr. para hash chain, ptr para lista de pág do nó-v, ptrs. para lista livre ou lista de pág a serem escritas, núm de processos a usarem a pág, sinc. e flags, e campos para HAT.
- O espaço de endereçamento consiste numa lista

ligada de mapeamentos para o processo (cada um deles um seg), uma pista para o último segmento que teve falha, flags, e o hat.

- `as` inclui operaçoes de alocaçao, remoção e duplicação para todo o espaço.
- Também inclui operações de mapeamento, proteção, e fault handlers para páginas.
- Cada segmento representa um mapeamento entre um objecto e uma região de endereçamento. Cada segmento inclui campos independentes do tipo (base, tamanho, ptr para as), um ptr para `seg_ops`, incluindo `duplicação`, `fault` e `faulta`, `setprot` e `checkprot`, `unmap`, `swapout`, `sync`, uma rotina `create`.
- Páginas anónimas não têm armazenamento permanente. São criadas quando se modifica uma pág com proteção `MAP_PRIVATE`, e são armazenadas em swap.
- `anon` exporta `dup`, `free`, `private` (cópia privada), `zero` e `getpage`.

- Existe também um *objecto anónimo* no sistema, representado pelo nó-v `NULL`. Páginas desse objecto são páginas anónimas.

O HAT

Operações dependentes do hw são isoladas em HAT. É acedido por uma interface que é chamada:

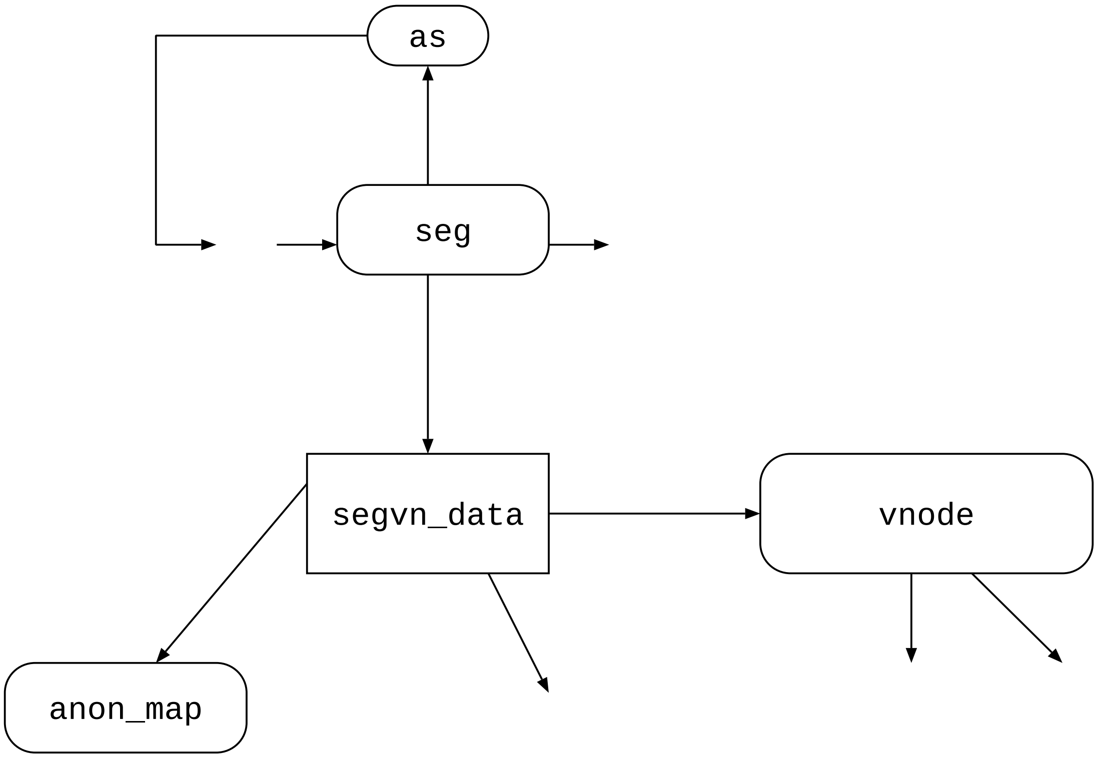
- HAT: alocação, dealocação, dup, swapin e swapout.
- conjunto de páginas: chgprt, unload, memload e devload (últimos para uma única página).
- todas as traduções de uma página: pageunload, pagesync.

Informação no HAT é redundante e altamente dependente do HW.

Drivers de Segmentos

Existem diferentes tipos de segmentos:

- `seg_vn`: endereços para ficheiros regulares ou anónimo.



Em data inclui proteções correntes e máximas, tipo de mapeamento (partilhado ou privado), ptr para nó-v, offset do segmento, ptr para mapa anónimo, e ptr para um array de protecções por página.

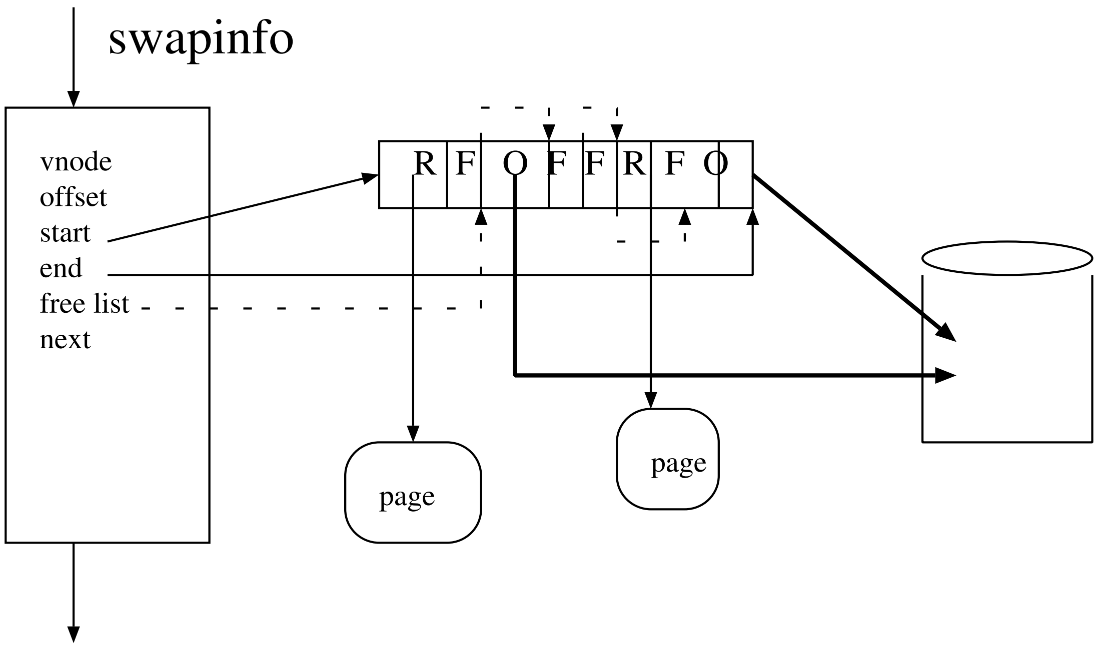
- `seg_map` permite fazer acessos do tipo `read` e `write`, substituindo a buffer cache. Existe apenas um no sustema e é uma versão optimizada do anterior.

- `seg_dev` mapeia devices como frame buffers, memória.
- `seg_kmem` mapeia regiões do kernel em memória alocada não dinâmicamente.
- `seg_kp` suporta lwps, incluindo red zones.

Swapping

- Rotina `swap_xlate` mapeia estruturas tipo anon e páginas outswapped.

- Dispositivos de swapping podem ser colocados e removidos dinâmicamente do sistema. Cada um deles tem uma estrutura tipo `swapinfo`.



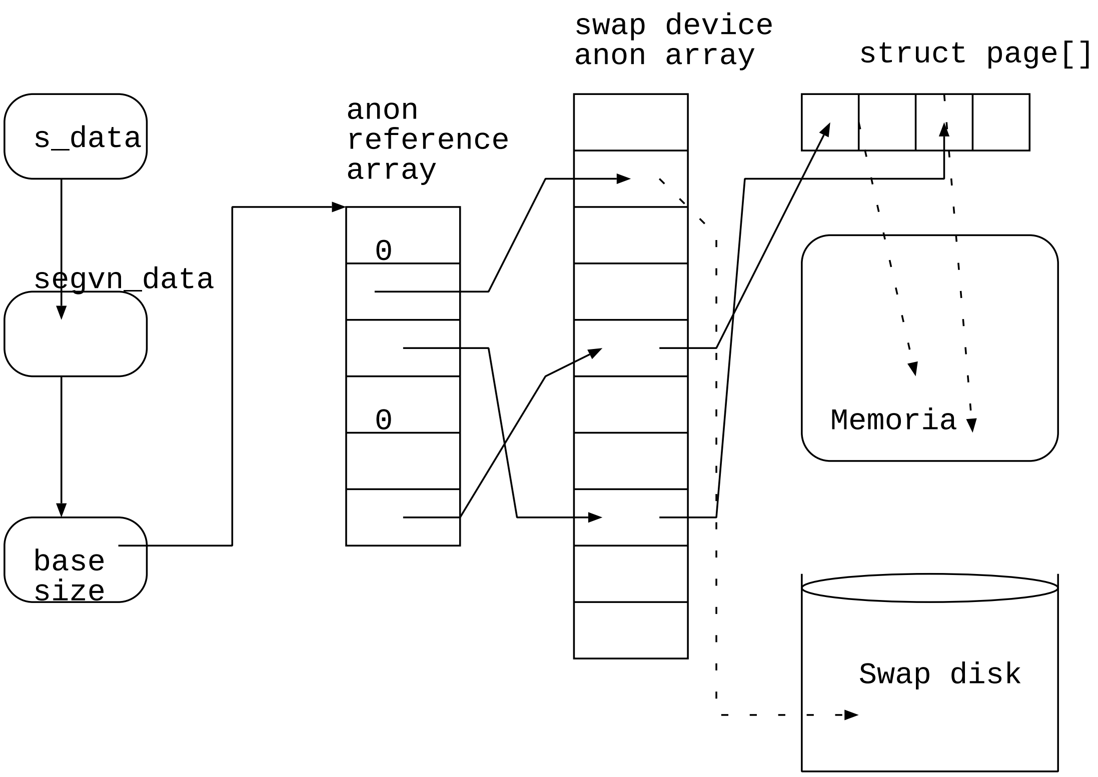
- Segmentos alocam todas as páginas de swap que vão precisar antes de começar.

Novo Mapeamento

- Regiões novas são mapeadas por `exec` ou por `mmap`.
- Operação inclui localizar o nó-v para o ficheiro, chamar `VOP_MAP` , verificar se não há sobreposição de mapeamentos, chamar `as_map` que aloca um `seg` e chama o `create` apropriado.
- Permissões não podem exceder aquelas com que o ficheiro foi o aberto.
- `exec` estabele mapeamentos privados para texto, dados e pilha. Pode também incluir mapeamentos para bibliotecas partilhadas.

Gestão de páginas Anónimas

- Páginas anónimas são criadas quando util. escreve para ficheiro com `MAP_PRIVATE`, ou em acesso a páginas partilhadas.



- anon array inclui núm de referências. Pág. são removidas quando núm. de ref. desce a zero.
- As estruturas de dados são criadas na primeira

escrita a pág. privada.