Sistemas Operacionais

Vítor Santos Costa

COPPE/Sistemas

Universidade Federal do Rio de Janeiro

O Kernel

O que é um Kernel?

- É um processo? Algo especial?
- Kernel é um programa (ou biblioteca) que corre directamente no HW;
- Implementa o modelo de processos e os outros serviços do sistema;
- Reside num arquivo, eg. `/vmunix`, `/unix`, `/boot/vmlinux` (RedHat), `/kernel`.
- `Bootstrapping` carregar o kernel de disco e inicializar o OS;
- Kernel inicializa dispositivos e fica em memória até shutdown.

Função do Kernel

- Esquema para a execução de programas
- Serviços como Entrada/Saída e gestão de arquivos
- Uma interface para esses serviços

Unix

Um dos kernels mais bem estudados:

- Originalmente de Bell Labs
- Contribuição de BSD
- Comercialização: SVR2 e SVR3
- SVR4 da AT&T
- OSF/1 baseado em Mach
- Solaris 2.x

Linux é um clone de Unix.

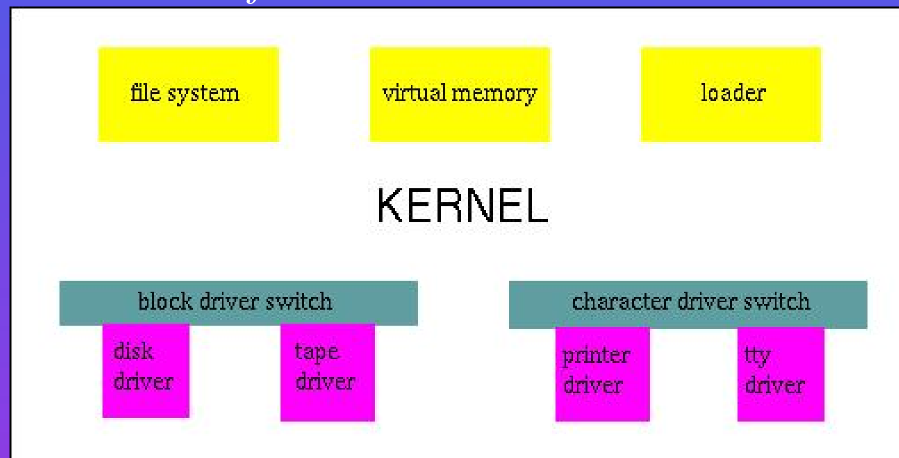
Unix

Características básicas:

- Multiprogramação: vários processos concorrentemente
- Cada processo roda em *máquina virtual*
- Espaço de endereçamento do processo é virtual
- Kernel gera memória, registos, CPU.
- Processos *bloqueiam* na falta de um recurso
- Suporta *time-slicing*

O Kernel de Unix

- Inicialmente, *small is beautiful*:



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

Funcionalidade do Kernel

4 formas de entrar no kernel:

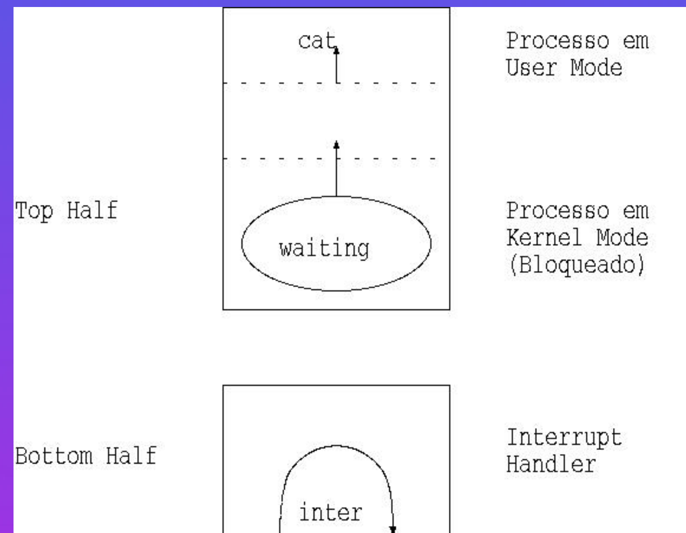
1. Explicitamente através da *System CALL Interface*.
2. *Excepção em HW* como divisão por 0, ou overflow de pilha, ou acesso a posição ilegal da memória. O Kernel sabe quem causou a excepção.
3. Interrupções de periféricos: não se sabe quem activou o dispositivo.
4. Processos correm sempre dentro do kernel e fazem serviços (*kernel daemons*): `nfsd`, `swapper`, `pagedaemon`.

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*.

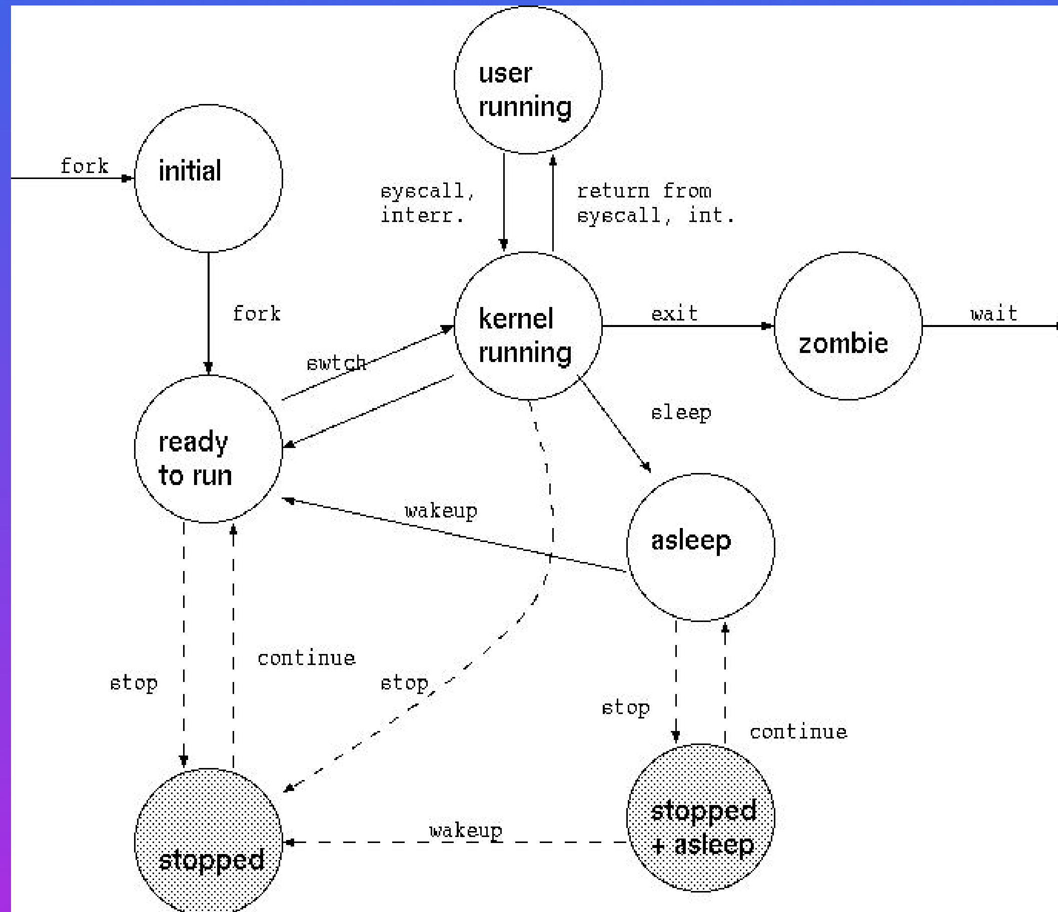
Modos de Execução e Contextos

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



- 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 modo-kernel; mapa de tradução de endereços.
- Credenciais: UIDs, GIDs.
- Variáveis de Ambiente
- Contexto HW: PC, SP, PSW (processor status word), mmem regs, FPU regs.

Credenciais de um Processo

- *UID* real e efectiva.
- *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.
- SYSV mantém saved UID e GID que sao restaurados por setuid.
- BSD suporta vários grupos por utilizador.

A Área-U

- PCB (process control block): armazena o contexto HW;
- pointer to `proc`;
- 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);
- Pilha em modo-kernel.

A Estrutura `proc`

- PID e SID (id da sessão).
- endereço da área u no kernel.
- estado do processo.
- ptrs para incluir o processo numa fila de escalonamento ou de “sleep”.
- *sleep channel* para processos bloqueados.
- prioridade de escalonamento.
- sinais que são aceites pelo processo.
- ponteiros para lista de processos activos, livres ou zombie.
- ponteiros para hierarquia de processos e para hash queue on PID.
- Gestão de memória e Flags Misc

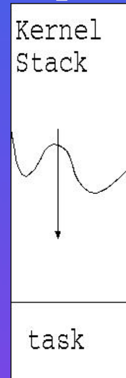
Estrutura de Processo Linux

Em `/usr/src/linux/include/linux/sched.h`

- ver `task_struct`
  - ⋆ Estado
  - ⋆ Prioridade e informação de escalonamento escalonamentos
  - ⋆ Formato de binários
  - ⋆ `pid`, `gid` e outras credenciais
  - ⋆ listas de processos
  - ⋆ gestão de sinais
  - ⋆ sistema de arquivos
  - ⋆ memória virtual

Linux: task_struct

- Alocação é feita com 2 páginas (960B para task_struct):



- Truque para encontrar task:

```
current = (struct task_struct *) (%esp & ~8191UL);
```

- Em x86 TSS é usado para guardar endereço da pilha kernel.
- Tabela de Hash através de PID;
- Solaris passa o endereço através da pilha.

FreeBSD: proc e user

- proc encontra-se em /usr/src/sys/sys/proc.h.
- user encontra-se em /usr/src/sys/sys/user.h.
- A partir de 4.4 BSD user só usada em fork():

```
struct  user {
        struct  pcb u_pcb;
        struct  sigacts u_sigacts;     /* p_sigacts points here (use it!) */
        struct  pstats u_stats;    /* p_stats points here (use it!) */
        /* Remaining fields only for core dump and/or ptrace--
         * not valid at other times!
         */
        struct  kinfo_proc u_kproc;        /* proc + eproc */
        struct  md_coredump u_md;        /* machine dependent glop */
};
```

Execução em Modo Kernel

Chamada de sistema:

1. Um wrapper chama a instrução `chmk` no VAX, `syscall` no MIPS, `trap` no MC68k, `LCALL` ou `int` no x86,. . . .
2. `syscall()` no kernel copia arguments e salta através da tabela `sysent` (`sys_call_table` em Linux).
3. No retorno copia valores de retorno em registo, restaura contexto HW, e regressa a user mode.

Implementação

- Linux: `arch/i386/kernel/entry.S`;
- FreeBSD: `i386/i386/trap.c`.

Gestão de interrupções

- BSD suport *interrupt priority level (ipl)*: 0-31.
- Quando saímos do Interrupt Handler verificamos se há alguma interrupção suspensa.
- 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()`.

Sincronização em Unix: Problemas

Interrupts podem acontecer a qualquer altura:

- Uniprocessadores: sincronização com
  - entrada: `x = splbio()`,
  - saída: `slpx(x)`.
- Multiprocessamento:
  - `locks`,
  - *semáforos*,
  - *rw_locks*.

Sincronização: Implementação

- em Linux `sleep_on()`, `wake_up_process()` e `schedule()` em `kernel/sched.c`.
- Não há níveis de interrupções: o sistema usa *handlers* que são executados depois da interrupção.
- Ideia semelhante: DPC de NT (deferred procedure call).
- Verificar `kernel/timer.c`
  - ⋆ Mecanismo sofisticado de expiração de timers (`run_timer_list` executa em tempo constante.
- *tasklets*: Kernels recentes permitem a bottom-half handlers de tipos diferentes executar em paralelo (SMP).
- APIC: divisão de interrupções entre CPUs.

Escalonamento em Unix

Partilha do CPU:

- Escalonador (ver `kernel/sched.c` em Linux e `kernel/kern_switch.c` e `i386/i386/swtch.s` em FreeBSD.
- Ideia é usar algoritmo round-robin com múltiplas filas de propridade.
- Processo mais prioritário entra no CPU mesmo antes do fim do quantum.
- Em Unix tradicional prioridade é função de *nice* e de *factor de uso*.
- Quando processo bloqueia no Kernel, no regresso recebe prioridade de Kernel *sleep priority* (ver `sys/param.h` em BSD).
- *sleep priorities* dependem da razão pq adormecemos.

Sinais em Unix

- Funcionalidade: comunicação entre processos, interrupts, e excepções.
- Cada sinal tem uma resposta default, geralmente terminação do processo.
- Sinal é colocado como um bit em `proc`, processo consulta `proc` antes de executar.

Sinais em Unix: Problemas

- Problema: sinais sobre processos adormecidos:
  - ⋆ se vai acordar cedo, o sinal pode bloquear.
  - ⋆ se vai ficar muito tempo, pode-se passar o interrupt.

BSD4.3 fornece `siginterrupt()` para controlar esta funcionalidade. `sigaction` e `SA_RESTART` têm o mesmo efeito.

Sinais em Unix: Implementação

- Linux:
  - ⋆ `kernel/signal.c`;
  - ⋆ em `arch/i386/kernel/entry.S` ver work_pending
  - ⋆ que leva a `arch/i386/kernel/signal.c`
- FreeBSD: `kern/kern_sig.c`

Novos Processos

- Em Unix `fork()` cria um novo processo:
  1. pai retorna de `fork()` com código do filho;
  2. filho retorna de `fork()` com `0`.
  3. Esta é a única diferença entre os 2
- Habitualmente filho executa `exec()` que faz overlay de um programa novo.
- Manter *fork()* e *exec()* separados:
  1. permite clones (client-server, prog. par);
  2. permite fazer setup antes de `exec()`;
  3. problemas de performance.

`fork()`

- `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 fila de escalonamento, retorna para o filho e para o pai.
- evitar cópia: copy-on-write (SYSV) e `vfork()` (BSD).
- ver `kernel/fork.c` em Linux e `kern/sys_fork.c` em FreeBSD.
- procurar PID: `get_pid()` em Linux.
- Linux retorna no filho, Unix no pai.

`exec()`

- `exec()` obtém o executável, verifica permissões, lê o cabeçalho, altera ID se SUID ou SGID, copia os argumentos de `exec()` e *env* para kernel space, aloca swap, liberta data e pilhas antigas, aloca mapas de endereço e inicializa-os, restaura *env* e argumentos, reinicializa os signal handlers, e inicializa contexto HW.
- Ver `fs/exec.c` em Linux, `kern/sys_exec.c` em FreeBSD.

exec() em Linux

1. flush_old_exec() limpa;
2. open_exec() procura o ficheiro;
3. search_binary_handler();
4. chama linux_binfmt $\rightarrow$ load_binary, eg load_elf_binary() em fs/binfmt_elf.c;
5. Outros formatos: aout, sh, misc, e em86.

init

em `init/main.c`, único processo que não resulta de `fork()`

```
init() {
 lock_kernel();
 do_basic_setup();
 /* Ok, we have completed the initial bootup, and we're essentially up
  * and running. Get rid of the initmem segments and start the user-mode stuff..  */
 free_initmem();
 unlock_kernel();

 if (open("/dev/console", O_RDWR, 0) < 0)
   printk("Warning: unable to open an initial console.\n");

 (void) dup(0);
 (void) dup(0);

 /* We try each of these until one succeeds.
  *
  * The Bourne shell can be used instead of init if we are
  * trying to recover a really broken machine. */
 if (execute_command)
  execve(execute_command,argv_init,envp_init);
 execve("/sbin/init",argv_init,envp_init);
 execve("/etc/init",argv_init,envp_init);
 execve("/bin/init",argv_init,envp_init);
 execve("/bin/sh",argv_init,envp_init);
 panic("No init found. Try passing init= option to kernel.");
}
```

Terminação de Processos

- `exit()` é chamada ou devido a um signal ou do próprio processo.
- processos podem comunicar por `kill()`:
  - ⋆ `-1`: todos
  - ⋆ `-PG`: grupo `PG`
  - ⋆ `0`: grupo do processo
  - ⋆ `PID`
- Em Linux implementado por `kill_something_info()`.

`exit()`: Algoritmo em Unix

- 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
- chama `swtch()`.

`exit()`: Implementação em Linux

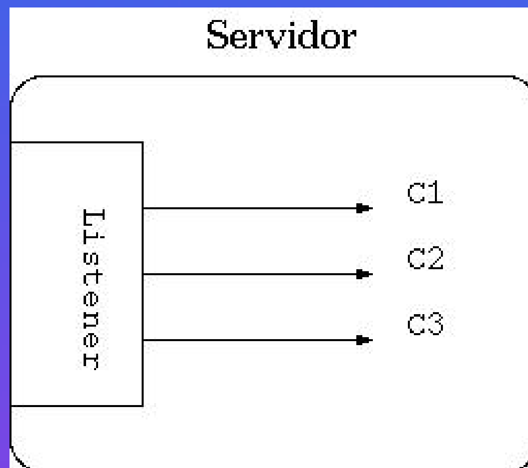
- Ver `kernel/exit.c` em Linux: `do_exit()` libera mm, `files`, `fs`, chama `exit_notify()`.
- `exit_notify()` chama `do_notify_parent()` e passa filhos para `init()`.
- `do_notify_parent` envia `SIGNAL` para pai e tenta acordá-lo.

Terminação do Processo: `wait()`

- `wait()` espera terminação de processos: retorna se já houver processos mortos, senão bloqueia. Em qq caso retorna `pid`, escreve o status do filho, liberta o `proc`.
- BSD4.4 fornece `wait4()` com info de recursos. POSIX fornece `waitpid()`. SVR4 tem `waitid` que fornece tudo.
- se processo morre depois do pai pertence a `init`.
- se processo morre antes do pai e este não chama `wait`, processo fica *zombie*. SVR4 permite usar `SA_NOCLDWAIT` sobre `SIGCHLD` para indicar que pai não vai esperar pelos filhos
- Implementado em `kernel/exit.c`: procura processo no estado `ZOMBIE` ou em `STOPPED` com exit code.

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

- Kernel Threads: baratos, não são associado com processos de utilizador, e têm a sua própria pilha.
- Úteis para AIO e interrupts. Equivalente ao *pagedaemon* e a *nfsd* em Unix.

Tipos de Threads: LWPs

- 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.
  - Kernel suporta sincronização (para blocking), context switching e escalonamento. Isso obriga a 2 *mode switches* atravessando *protection boundary*.

Tipos de Threads: User Threads

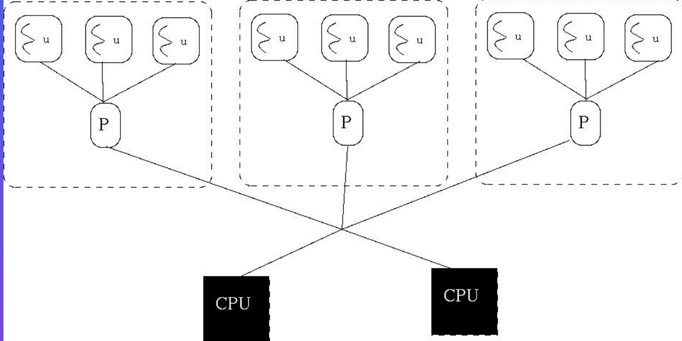
- User Threads: implementados por bibliotecas (C-threads ou pthreads).
  - Muito eficientes:

	Criar	Sincronizar Semáforos
UT	52	66
LWP	350	390
P	1700	200

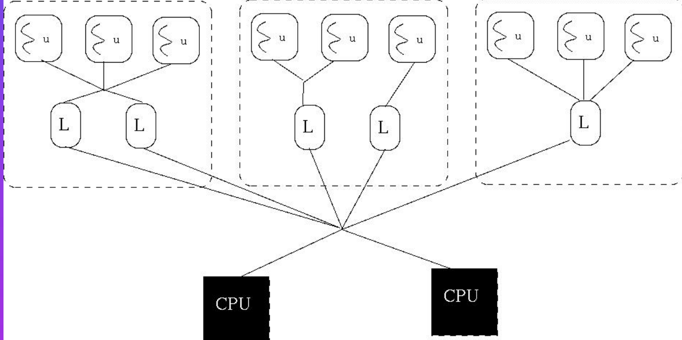
  - não têm paralelismo.
  - escalonamento é feito pela aplicação.
  - AIO permite não bloquear thread, mas à custa de complexidade de programação.
  - Ideais para aplicações gráficas.
- Problema, separação entre UT e LWP:
  - kernel não pode saber que LWP tem os melhores threads;
  - UTs podem perder LWPs, não suportam paralelismo.

UTs e LWPs

UTs:



UTs+LWPs:



Suporte a LWPs no Kernel: `fork()`

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

1. Segunda melhor para `exec()`, mas problemas com bib. que tenham os seus próprios 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.
3. Registrar *fork handler* que são executados antes e depois de `fork()`.

Outras Chamadas de Sistema

- Vários LWPs podem aceder ao mesmo fd (um lê, outro faz fseek). Soluções:
  - ⋆ aplicação resolve o problema;
  - ⋆ kernel suporta random IO atómico (`pread` e `pwrite`)
- cwd e credenciais únicos no processo
- gestão do mapa de memória (vários `brk()` ao mesmo tempo).

Sinais

- Quem recebe sinais?
  - ⋆ Todos os LWPs (`^Z`);
  - ⋆ Qualquer um
  - ⋆ Master
  - ⋆ Heuristicas
  - ⋆ Novo LWP
- Sinais como `SIGSEGV` devem interromper LWP responsavel.
- `SIGINT` é interrupção: não pertence a um thread
- Signal Handler
  - ⋆ geral: não tem overhead
  - ⋆ privado: mais versátil
- Signal masks devem ser privados para proteger regiões críticas.

Visibilidade e Pilhas

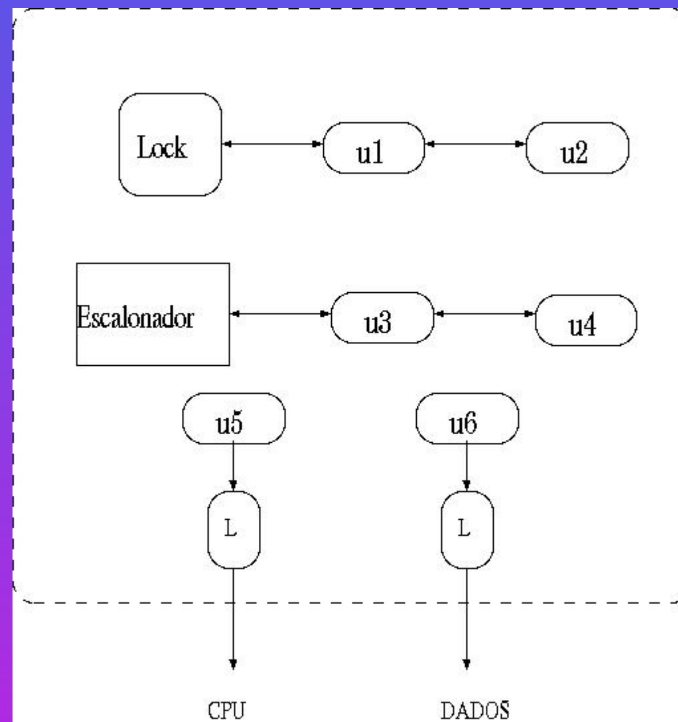
- Maioria das packages não permite a LWPs serem visíveis fora do processo
- Permitir sinais para comunicação dentro do processo
- Gestão das pilhas:
  - SEGV causada por overflow em user thread
  - kernel não deve ser invocado
  - SEGV é tratado por biblioteca
  - note que SEGV handler não pode usar pilha. Porquê?

P-Threads

- criar: `pthread_create(THREAD, ATTR,ROUTINE,ARG)`
- terminar: ímplicito ou por `pthread_exit(RETVAL)`.
- atributos: `detachstate`, `schedpolicy`, `schedparam` (prioridade), `inheritsched`, `scope` (não em Linux).
- cancelar outro thread: `pthread_cancel(TH)`
- Primitivas de sincronização: *mutexes*, *variáveis de condição*, *semáforos*, *read-write locks*.
- handlers: cleanup, `at_fork()`.
- Variáveis privadas a threads: usa `TSD` indexada por chave.
- Processamento e envio de sinais.
- `pthread_join(TH,THREAD_JOIN)`.

Implementação de Bibliotecas

- Implementação:
  - LWP para UT;
  - Multiplos UT num LWP;
  - Permita UT ligados e não ligados. Pode favorecer bound threads



Solaris: Kernel Level Threads

- Kernel Threads são usados para actividade assíncrona (*callouts*, STREAMs, escrita no disco) 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.
- Kernel organizado como conj. de KTs: alguns LWPs, outros no kernel.
- KTs são *preemptible*.
- Primitivas de sincronização: semáforos, condições, etc, tentam impedir *inversão de prioridades*.

Solaris: LWPs

- Cada LWP é associado a um KT durante a sua vida.
- Em `lwp`:
  - valores de registos usuário.
  - argumentos e resultados para `syscalls`.
  - info. para signals.
  - alarmes em tempo virtual; tempo de util. e CPU; outros recursos.
  - ptr. para KT.
  - ptr. para `proc`
- `LWP` é swappable, logo máscaras têm que estar em KT. No SPARC **g7** refere `lwp`.
- Sincronização como para KT: bloqueantes ou não.
- Signal handlers são comuns ao processo mas máscaras pertencem ao LWP

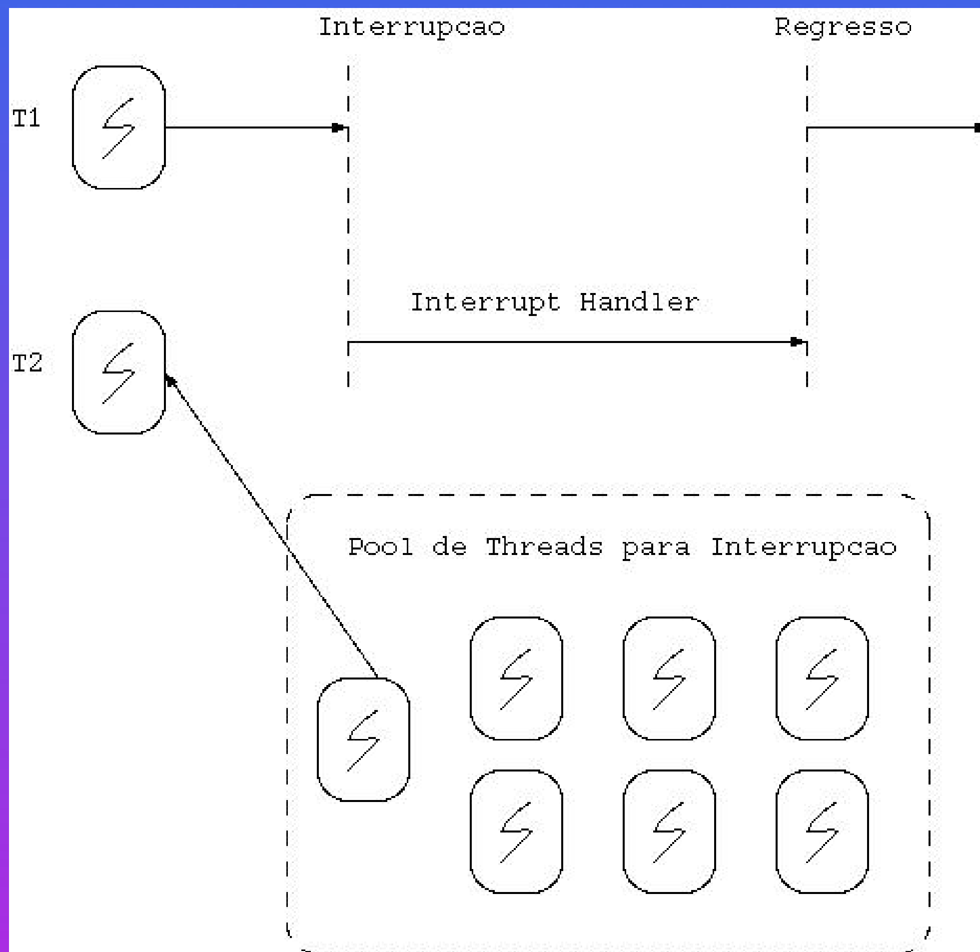
Solaris: User Threadss

- UT são implementados pela *threads library*.
- Podem ser associados a LWPs ou não
- Implementação
  - Thread ID.
  - saved register state
  - pilha de utilizador.
  - máscara de sinais.
  - prioridade
  - armazenamento local (`errno`).
- Versões recentes suportam PThreads.

Interrupções e Interrupções

- Kernels tradicionais usam *ipl* para proteção de recursos partilhados em interrupções
- Acesso a recurso partilhado requer mexer em IPL.
- Solaris tb usa mutex e semáforos
- Interrupções bloqueiam apenas em situações excepcionais
  - eg, tentando adquirir um mutex que protege a *sleep queue*
- custo de criar KT é muito alto:
  - pool de threads, inicializados parcialmente
  - só completamente inicializados se bloquearem
  - Um por cada nível de interrupção exigindo 8k de espaço.
  - Pode gastar muita memória
- Durante a sua execução, interrupts threads prendem o thread que foi interrompido ao processador.

Solaris: Interrupts



Solaris: Chamadas de Sistema

Chamadas de sistema:

- `fork` duplica todos os LWPs
  - ⋆ os threads em `syscall` recebem `EINTR`.
- `fork1` só um thread.
  - ⋆ util se logo antes de `exec()`
- `pread` e `pwrite` fazem `fseek+op`.
- Não existe `preadv` e `pwritev`
- programadores podem começar com threads e depois usar LWPs.

Mach: Threads

Mach suporta:

- Task: objecto com:
  - espaço de endereçamento
  - 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`).

C-Threads

- Fornece interface para usuário
- Mutexes e variáveis de condição para sincronização
- Usuário pode escolher
  - `coroutine`
    - precisa de `cthread_yield()` para mudar voluntáriamente
    - senão muda de contexto apenas em sincronização
  - `threads`: default
  - `tasks`
    - Usa VM para comunicar

Mach: Tasks

Task contém:

- ptr. para mapa de endereços (VM).
- ptr. para lista de threads na task.
- ptr. para o *processor set* da task.
- ptr. para `utask` (compatibilidade com Unix).
- portas e mais info IPC.

Mach: Threads

Thread contém:

- Links para fila de escalonador ou de wait.
- ptr. para task e processor set.
- links para lista de threads da task e proc. Set.
- ptr. para PCB com contexto.
- ptr. para pilha de kernel.
- Estado de escalonamento (pronto, suspenso, bloqueado,...)
- Info. de escalonomento (PRIO, policy, uso de CPU).
- ptr para `uthread` e `utask`
- IPC.

Digital Unix

Baseado em Mach 2.5:

- Externamente: Unix.
- Internamente: Mach.
- `área-u` é substituida por:
  - `utask`: vnode (cwd, root), `proc`, signal handlers, open file descriptors, *cmask*, recursos.
  - `uthread`: registos, travessia de caminhos, sinais, handlers para threads.
- macros permitem converter de `u-area` para as novas regiões.
- `proc` esvaziada por `task` e `thread`: muitos campos não são usados.
- fork cria um novo thread.

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

- Problema:

```
sys_call_1(arg1) {
  ....
  save arg1 e info de estado;
  thread_block();
  f2(arg1);
  return
}

f2 (arg1)
{
   ....
}
```

Mach: Continuações

Mach 3.0 usa *continuações*, uma função a executar quando o thread bloqueia:

- Thread primeiro guarda algumas variáveis e devolve a sua pilha.

```
sys_call_1(arg1) {
....
save arg1 e info de estado;
thread_block(f2);
/* não executado */
}
```

- Quando é reexecutado, usa variáveis e a continuação para recuperar estado.

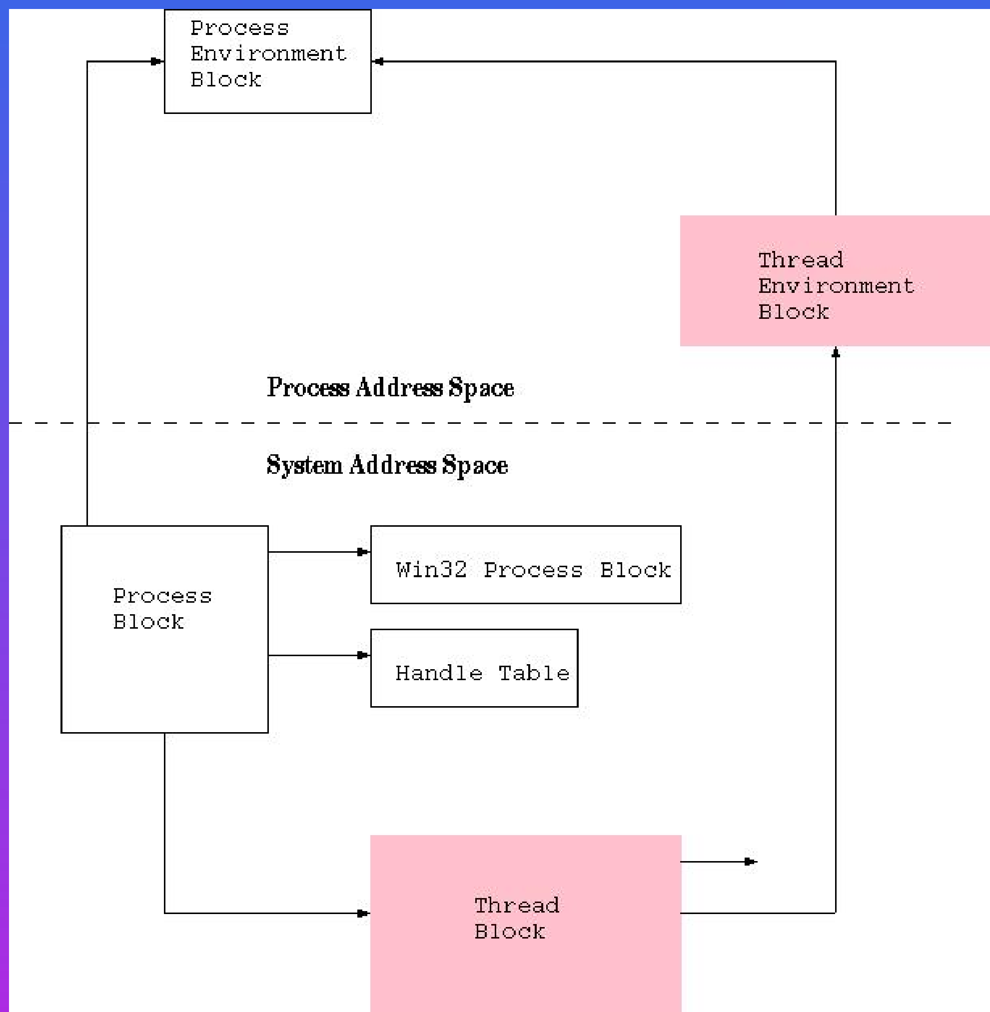
```
f2() {
recupera arg1 e info de estado;
...
thread_syscall_return()status;
}
```

- 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).
- Em mensagens kernel pode transferir pilha directamente:
  - ⋆ evita TLB e cache misses.
  - ⋆ cliente chama `mach_msg()` numa *port* e espera que servidor responda com `mach_msg()`. Se servidor não pronto, mensagem é colocada numa fila.
  - ⋆ Se o emissor tem o receptor à espera, emissor passa-lhe a pilha
  - ⋆ emissor depois bloqueia com `mach_msg_continue()`
  - ⋆ Aí, receptor execute imediatamente usando a pilha do emissor onde já está a mensagem!
- Ideal para hot-spots (típicos em interface pequena, como num micro-kernel).

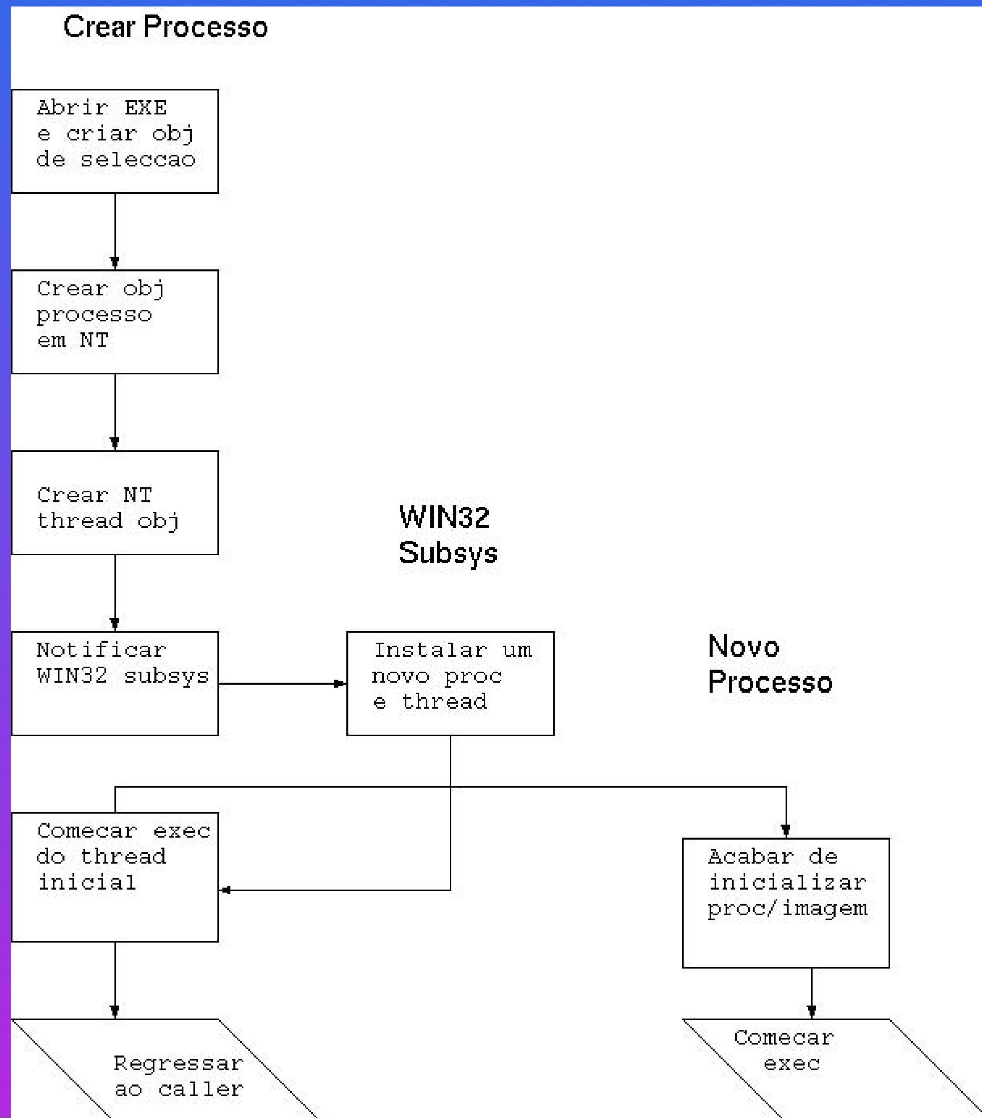
Windows NT: Processos



Windows NT: Executive Process Block

- Bloco `KProcess`: dispatcher object, ptr. para process pages, KTHREADs para o processo, prioridade base, quantum, afinidade, tempos em kernel e usuário.
- PID: ID do processo e pai, nome da imagem, “window station”.
- Bloco para Quota.
- Descriptores de Espaço de Memória Virtual.
- Info. sobre conjunto de Trabalho.
- Info sobre memória virtual.
- Porta para Excepções.
- Porta para Debugging.
- Token de Acesso (profile de segurança).
- Tabela de handles para objectos.
- W32Process e PEB.

Windows NT: CreateProcess



Windows NT: o PEB

- Sempre mapeado no endereço `0x7FFDF000`.
- Info usada pelo carregador, gestor de heap, e outros DLLs Win32.
- Inclui:
  - ⋆ Endereço base da imagem;
  - ⋆ Lista de módulos;
  - ⋆ Dados locais a threads;
  - ⋆ Time-out de secção critica;
  - ⋆ Número de heaps;
  - ⋆ Tamanho da heap;
  - ⋆ Ptr. para heap;
  - ⋆ handle partilhada para GDI;
  - ⋆ versão do OS e da imagem;
  - ⋆ Afinidade.

Linux: `clone`

- Ideia vem do Plan9.
- Processos podem partilhar diferentes recursos: `VM`, `FS`, `FILES`, `SIGHAND`, `PID`, `PTRACE`, `VFORK`, `PARENT`, `THREAD` (thread group).
- Processos e threads são a mesma coisa.
- Diferença mais importante:

  no process switch, mudança de TLB não é necessário se têm o mesmo `cr3`.
- Kernel:

  2.0: BGL.

  2.2: SMP threaded, mas muito código sequencial.

  2.4: SMP threaded, VM, rede, file systems são paralelos.
- New Generation Kernel Threading: http://www.opengroup.org/rtforum/jan2002/slides/linux/abt.pdf

Scheduler Activations

Integrar UTs e Kernel:

- Kernel aloca CPU(s);
- Biblioteca escalona.
- Biblioteca informa sobre eventos que afectam alocação: pede mais CPUs, libertar CPU.
- Kernel controla alocação e pode retirar CPUs.
- Mas, quando biblioteca tem CPU é ela *quem escolhe que UT corre lá*.
- Kernel deve informar biblioteca sobre mudanças.

Ideia veio de Anderson

Scheduler Activations: Abstrações

Upcall: Kernel chama biblioteca;

Scheduler Activation: contexto que pode ser usado para correr um UT (semelhante a LWP).

- Quando kernel faz upcall passa ou retira activação para a biblioteca.
- Kernel não faz timeslice sobre activações.
- Bloqueio:

1. kernel cria uma nova activação e faz upcall.
2. bibl. guarda activação antiga, liberta-a, informa o kernel.
3. bibl. escalona novo UT.
4. Quando operação conclui, nova upcall do kernel: nova activação. Pode dar CPU, ou remover uma activação.

Esquema extremamente rápido.

Scheduler Activations: Para Ler

- Paper original: http://www.cs.washington.edu/homes/tom/
- NetBSD: http://web.mit.edu/nathanw/www/usenix/
- FreeBSD: http://www.freebsd.org/kse/
- Mach 3.0:

Sinais Unix: Semântica

Mecanismo Básico de Comunicação em Unix:

- Originalmente não-fiável e falha:
- BSD4.2 corrigiu e melhorou muitas coisas.
- System V evoluiu diferentemente.
- POSIX.1 implementou standard.

Sinais Unix: Implementação

- 15 sinais, depois 31.
- Mecanismo:
  1. Geração;
  2. Pendente;
  3. Entrega
- acções default: *abort*, *exit*, *ignore*, *stop* e *continue*.
- processo pode redefinir handlers e bloquear, mesmo temporáriamente.
- qq acção tem que ser efectuada pelo próprio processo, incluindo *exit*.

Sinais: Implementação

Ideias Básicas

- Processo chama *issig()* para ver se tem sinal:
  1. antes de retornar para user-mode (Linux testa `sigpending` em `arch/i386/kernel/entry.S`);
  2. antes de bloquear num evento interruptivel `signal_pending()` de `include/linux/sched.h`;
  3. logo depois de acordar de um evento interruptivel.
- Se tiver, kernel chama `psig()` que despacha o sinal:
  - ⋆ termina o processo e gera `core`
  - ⋆ ou chama `sendsig()` para chamar código do usuário.

Sinais: Problemas

Sinais em SVR2 e antes não eram fiáveis:

- Signal Handlers não são persistentes e operação é reset para default *antes* de chamar o handler: *corridas*.
- Como `área-u` de outro processo é inacessível, tem que acordar o outro sinal.
- Sinais não podem ser bloqueados temporáriamente.
- BSD4.2:
  1. Handlers são persistentes;
  2. Sinais podem ser mascarados temporáriamente;
  3. Processos adormecidos não têm que ser acordados.
  4. `sigpause()` espera por um sinal.

Sinais: Sessões e Grupos

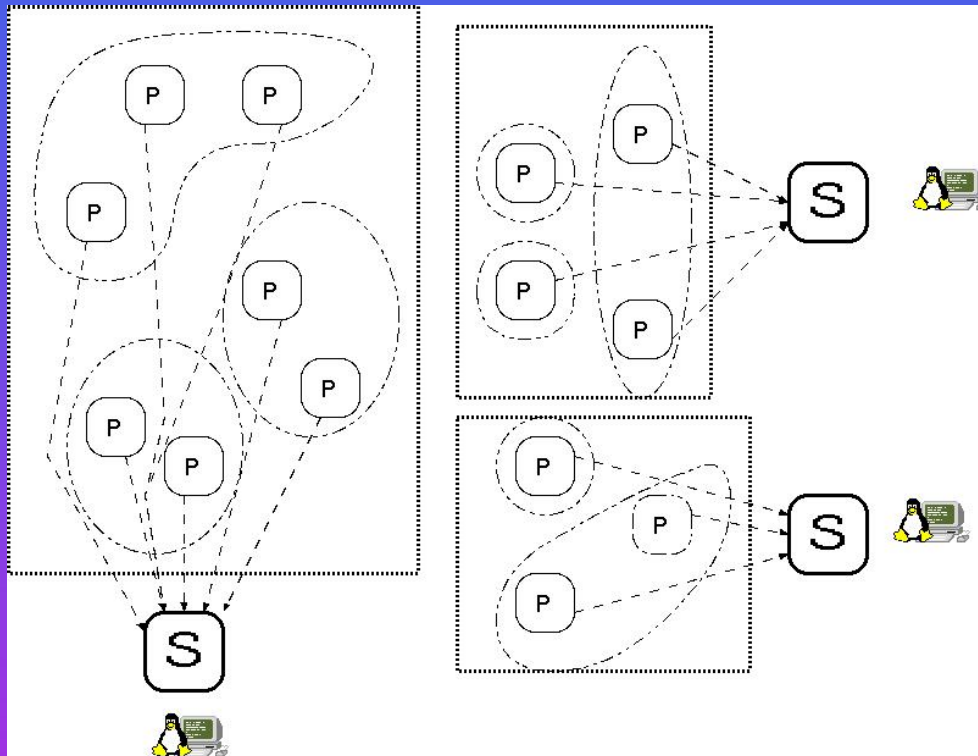
Grupos (BSD) estão associados a uma tarefa:

- cada processo tem um GID (`setgrp` muda);
- estrutura `tty` sabe que grupo é foreground, ie, controla o terminal.
- Se processo com GID de 0 abre terminal, terminal passa ser o *terminal de contrôle* para o processo, e processo junta-se ao grupo do terminal. Se terminal não é controlado, processo passa ser *leader de grupo*.
- Processo em foreground tem acesso irrestrito ao terminal.
- Se processo em background tenta ler do terminal, processos no seu grupo recebem `SIGTTIN`; escritas são permitas por default, ou `SIGTTOU`.
- o `ioctl` `TIOCSPGRP` pode mudar o dono do terminal: usado pela shell para passar de foreground para background.
- Quando todos os processos fecham o terminal, terminal é disassociado e perde o dono.
- BSD pode reinicializar terminal.

Grupos e Sessões

- Problemas com Grupos:
  - ⋆ Não há sessão de logins (*sessions*);
  - ⋆ nenhum processo é responsável pelo terminal;
  - ⋆ processo pode colocar o grupo de controle como inexistente;
  - ⋆ incompatível com SYSV.
- Sessões: o leader da sessão é responsável por controlar o terminal.

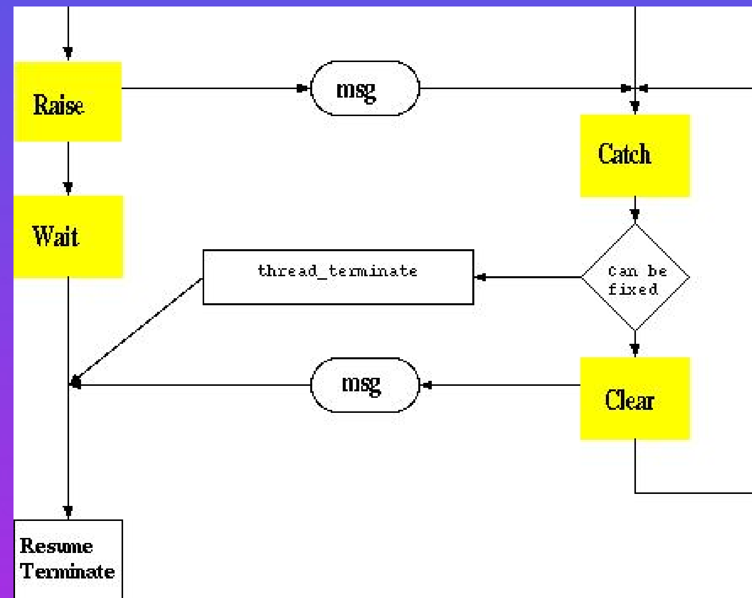
Sinais: sessões



Mach: Excepções

Handler pode executar em diferente contexto (multi-threading):

- Vítima: causa excepção e espera resposta;
- *Handler* processa resposta.



Mach: Portas

Porta é uma *fila protegida da mensagens*:

- Várias tarefas podem ter direitos de envio;
- Apenas uma tem direitos de recepção.
- Portas de excepção podem estar associadas com threads e com tasks:
  - ⋆ Processamento de erros estão associados a threads.
  - ⋆ mas, novo thread tem porta de excepção `NULL`.
  - ⋆ Debugger regista-se como receptor para a porta de excepção de uma task.
  - ⋆ Excepções são enviadas para o thread error handler primeiro, depois para o task error handler.

Problemas de Escalonamento

Unix é *time-sharing*, ilusão de múltiplos processos concorrentes:

- *Estratégia (Policy)*: regras usadas para decidir que processo colocar e quando mudar;
- *Implementação*: estruturas de dados e algoritmos usados na implementação do sistema

Objectivos conflituantes:

- Resposta rápida para processos interactivos;
- *throughput* alto para processos background;
- evitar "starvation"

Implementação exige *context switch*, uma operação cara.

Context Switch

1. Guardar registos correntes no `PCB`;
2. Ler `PCB` do novo registo corrente;
3. Tarefas específicas da arquitectura:
   - Flush de caches de dados, instruções, ou TLB;
   - Prejudica o pipeline e reduz localidade.
   - Tb fazer flush do pipeline.
4. Custos influenciam escolha da melhor estratégia.

Ver `__switch_to` em `arch/i386/process.c` e `kernel/sched.c` para Linux.
Ver `cpu_switch` em `i386/i386/swtch.s` para FreeBSD.

Clocks

OS interrompido `HZ` ticks por segundo:

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

Algumas tarefas só são processadas no *major tick*.

Em Linux `do_timer_interrupt()` `(arch/i386/kernel/time.c)` → `do_timer()` `(kernel/timer.c)` → `mark_bh()` `(include/kernel/interrupt.h)` → `tasklet_action()` `(kernel/softirq.c)`

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:

1. "tempo até disparar" em BSD;
2. ringlist em Linux: ver `run_timer_list()` em `kernel/timer.c`.

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()`.
- POSIX fornece `nanosleep()` com precisão de nano-segundos.
- Note que processo só responde ao sinal quando é escalonado, o que afecta precisão.

Ver `kernel/timer.c` e `kernel/itimer.c` em Linux.

Política de Escalonamento em Unix

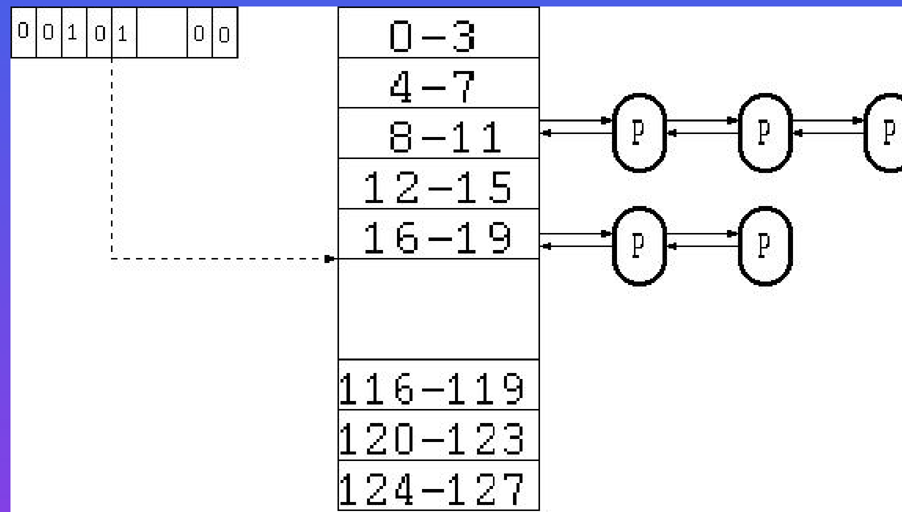
Três tipos de aplicações: interactivas, `batch` e real-time.

- 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.
- $\texttt{p_usrpri} = \text{PUSER} + (\texttt{p_cpu}/4) + (2 \times \texttt{p_nice})$
- `p_cpu` decai por um factor de 1/2 em SVR3 e $(2 \times \texttt{load_average})/(2 \times \texttt{load_average} + 1)$ em BSD, activado de segundo a segundo por um callout. `load_average` é o número médio de processos executáveis no último segundo.
- BSD previne "starvation": factor dependente do load evita que prioridades aumentem quando a load aumenta.

Implementação do Escalonador BSD



- 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.

Implementação do Escalonador BSD

- 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 em BSD

Problemas do Escalonador BSD:

- 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 aplicações em tempo real.
- 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.

Princípios do Escalonador de SVR4

Ideias principais:

- Fornecidas duas classes: time-sharing e tempo-real.
- Processamento independentes de classe para:
  - ⋆ mudança de contexto;
  - ⋆ manipulação da fila de processos;
  - ⋆ “preemption”.
- Interface para funções com herança e prioridades.

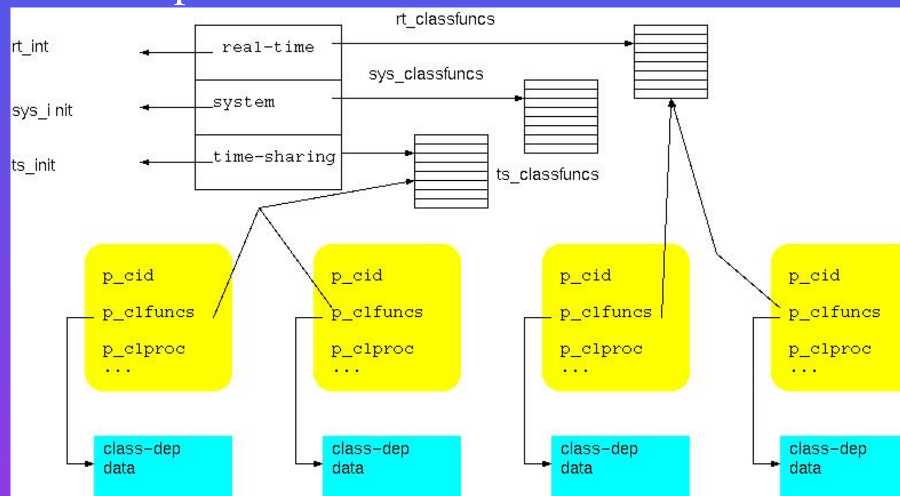
Processamento Independente de Classe

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; 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 de classe com `priocntl()`
- `proc` inclui ptrs. para id da classe, funções da classe, e estruturas de dados privadas.

Processamento Dep de Classe: Interface

- CL_TICK é chamada do relógio: time slice, recomputa prio, expiração do quantum.
- CL_FORK inicializa.
- CL_FORKRET inicializa runrun permitindo ao filho correr primeiro.
- CL_ENTERCLASS e CL_EXITCLASS são chamadas ao entrar e sair de 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 são divididas entre:

- 0-59 para time-sharing;
- 60-99 para system;
- 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:
  - `ts_timeleft`: tempo para terminar o quantum;
  - `ts_cpupri`, a parte de sistema;
  - `ts_upri`, parte de usuário (`nice`);
  - `ts_umdpri`: prioridade em modo user é

$$max(0, min(59, \texttt{ts_cpupri} + \texttt{ts_upri}))$$

  - `ts_dispwait`: tempo de relógio desde o início do quantum.
- Em modo kernel prioridade é determinada pela condição de sleep, depois é restaurada de `ts_umdpri`.

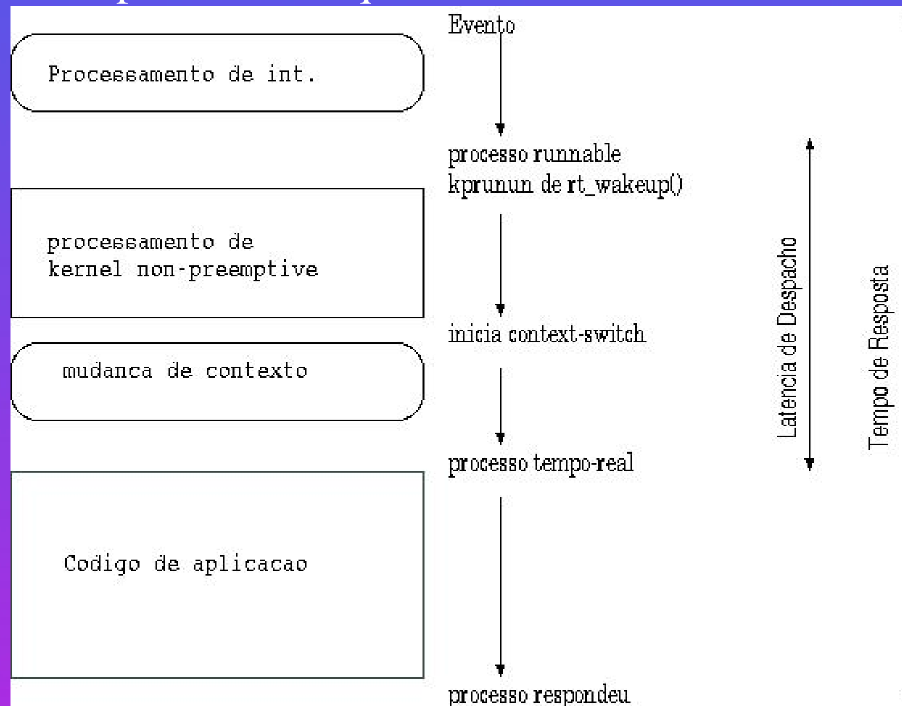
Time-Sharing em SVR4

	glbpri	quant	tqexp	slpret	mxwt	lwait
0	0	100	0	10	5	10
1	1	100	0	11	5	10
...	...	...	...	...	...	...
15	15	80	7	25	5	25
...	...	...	...	...	...	...
40	40	20	30	50	5	50
...	...	...	...	...	...	...
59	59	10	49	59	5	59

- `ts_globpri`: prioridade global;
- `ts_quantum`: quantum;
- `ts_tqexp`: `ts_cpupri` depois de quantum;
- `ts_tqexp`: `ts_cpupri` depois de sleep;
- `ts_maxwait`: número de segundos para esperar fim de quantum antes de usar `ts_lwait`.
- `ts_lwait`: use em vez de `ts_tqexp` se o processo demorar mais do que `ts_maxwait` para gastar quantum.

Processos de Tempo-Real em SVR4

- Exigem tempo de latência e tempo de resposta limitadas.
- prioridade maior do que processos em modo kernel.
- Escalonamento com prioridade e quantum fixos.



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.
- Ajustes podem ser necessários para manter equilibrio e evitar prejudicar processos interactivos com computação.
- Permite definir uma nova classe sem mexer no código do kernel.
- mas, `priocntl()` é restrito ao superuser.
- real-time não é deadline-driven.
- Difícil encontrar prioridades certas (http://www.cs.columbia.edu/~nieh/: escrita, 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 mecanismo de escalonamento diferente:

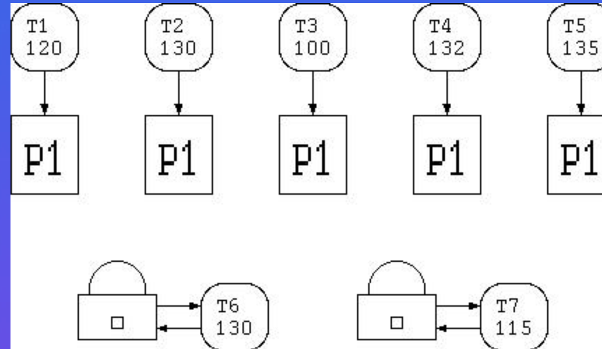
- Kernel é “preemptive”.
- Threads de interrupt permitem evitar `ipl`
- Suporte a multiprocessamento.
- Evitar escalonamento escondido
- Herança de prioridades
- Turnstiles.

Escalonamento para MP em Solaris

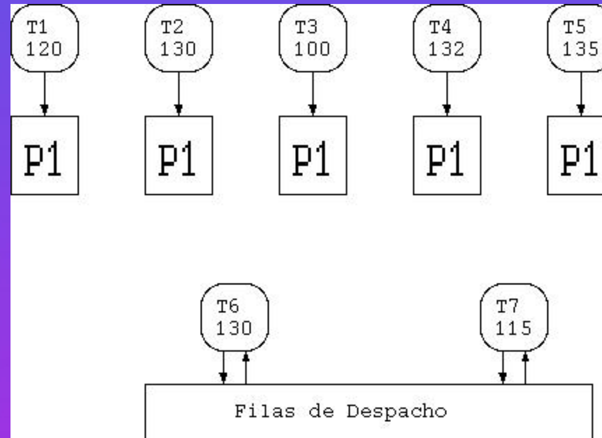
Suporte a Multiprocessadores inclui:

- Única fila de despacho.
- Threads podem ser restritos a um processador.
- Processadores podem enviar cross-processor interrupts.
- Cada processador mantém:
  - `cpu_thread` executando;
  - `cpu_dispthread`, o último thread executado;
  - `cpu_idle` thread;
  - `cpu_runrun`, `cpu_kprunun`; `cpu_chosen_level`, prioridade do thread que vai tomar o processador.
- Se P_i tem um processo com maior prioridade que P_j, coloca o seu `chosen_level` e envia um IPI para P_j.

Uso de `chosen_level`



T6 e T7 acordam:



Garante que T7 fica na fila, mesmo que outro CPU veja que P3 está a correr com prioridade 100.

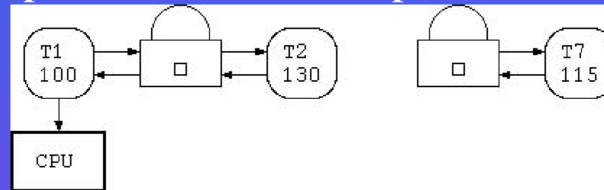
Escalonamento Escondido

O kernel faz trabalho assíncrono, sem considerar a prioridade das threads que fizeram a chamada original:

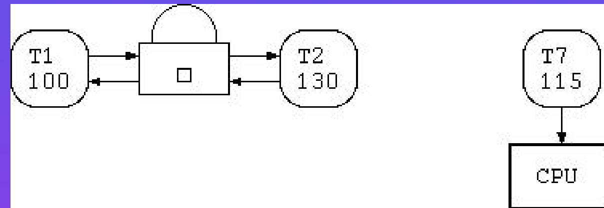
- Kernel pode verificar pedidos em STREAMS, que são servidos pelo e com a prioridade do processo actual em modo kernel.
  - ⋆ Ideia: STREAMs é feito em modo kernel, e abaixo de tempo real.
- Problema: pedidos de STREAMs feitos por processos de tempo-real?
- Callouts têm o mesmo problema, por principio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

Inversão de Prioridades

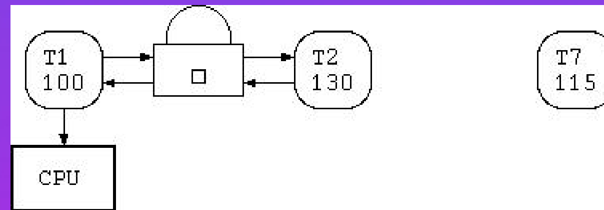
Thread de baixa prioridade pode ser necessário para activar thread de alta prioridade:



Quando T3 acorda:



Solução correcta:



O problema pode ser recursivo!

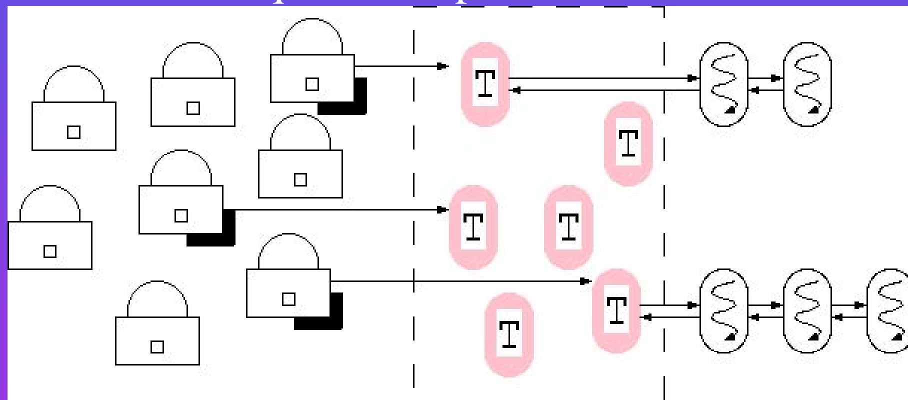
Inversão de Prioridades: Solução

- Herança de prioridade: threads têm *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.
- readers-writers: Solaris usa *owner-of-record*, primeiro thread a ler o objecto.
- Herança de prioridades reduz tempo de espera, mas não garante TR, nem evita que cadeias de bloqueamento cresçam.

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 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:
  - ⋆ Ignora overhead de context switches.
- Prioridade-base por task + factor de uso por thread, decaindo a $5/8$ por segundo inactivo.
- Cálculos são feitos pelo thread qdo acorda, e pelo relógio. Um thread interno re-computa prioridades de 2 em 2 segundos.
- Um thread corre até ao fim do quantum. Cede CPU com thread de $>$ prio.
- *handoff scheduling*: thread pode passar controle a outro:
  - ⋆ Útil para IPC.

Paralelismo em Mach

- Mach não usa IPIs: atraso prejudica RT, não time-sharing.
- Utilizadores podem criar *conjuntos de processadores*. Um servidor determina a alocação.
- Threads podem ser forçados a correr num CPU: útil para servidores sequenciais, ie de UNIX.
- É possível dedicar um conjunto de CPUs a uma task: *gang scheduling*.
  - útil para barreiras pq nenhum thread se atrasa;
  - e aplicaçãoes fine-grained, pq podem atrasar num thread suspenso.
- Cada CPU tem uma fila local, e existe fila para o conjunto de trabalho.
- Filas locais são vistas primeiro.

Escalonamento em True64 Unix

- `sched_setscheduler`: 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:
  - Time-sharing entre 0 e 29.
  - Máximo é 63.
  - Para ir acima de 19 processo precisa de *superuser*.
- `sched_setparam` muda prioridades de processo FIFO e round-robin;
- `sched_yield` cede o resto do quantum para outro processo com a mesma prio.

Escalo. Paralelo em True64 Unix

- Objectivos:
  - optimizar mudanças de contexto;
  - optimizar utilização de cache;
  - evitar problemas de luta por recursos quando vários processos são acordados da mesma fila global.
- Cada CPU tem uma fila local, e existe fila global.
- Escalonador tenta manter as filas equilibradas.
- 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.

Escalonamento em Linux

Implementado por `schedule()` em `kernel/sched.c`:

- Classes de escalonamento semelhantes a True64 Unix.
- Usa `goodness()` para estimar em que ponto o processo precisa do CPU:
  1. em `YIELD`, retorna -1;
  2. `RT` ou `FIFO`: retorna $1000 + \texttt{rt_priority}$;
  3. `OTHER`: se $\texttt{p} \rightarrow \texttt{counter} = 0$, dá 0;
  4. começa com $\texttt{p} \rightarrow \texttt{counter}$;
  5. dá $+$`PROC_CHANGE_PENALTY` se tiver corrido no mesmo CPU (15 no x86);
  6. dá $+1$ se tiver mesmo mm;
  7. dá $+20$ e subtrai $\texttt{p} \rightarrow \texttt{nice}$.

Escalonamento em Linux

- `schedule()` percorre a lista dos processos activos e escolhe aquele com maior goodness.
- `p` $\rightarrow$ `counter` é ajustado em:
  1. se nenhum processo escalonável tiver quanta, usando `p` $\rightarrow$ `nice`.
  2. timer decrementa `p` $\rightarrow$ `counter` e se 0 coloca `p` $\rightarrow$ `need_resched` a 1 (`update_process_times()`).
  3. pai divide com filho em `fork()`;
  4. pai recupera `p` $\rightarrow$ `counter` do filho em `exit()`;
  5. processo de tempo-real força recomputação se tiver interrompido processo time-sharing `p` com `p` $\rightarrow$ `counter` $== 0$.
- `wake_up_process()`: processos que acordam podem forçar reescalonamento se tiverem maior goodness que processo no CPU currente; complexo em SMP.

Escalonamento em Linux: Discussão

- Linux pode definir interfaces em source: aumenta eficiência
- Problemas:
  1. Suporte a muitos threads: http://www-4.ibm.com/software/developer/library/java2/
  2. Escalabilidade para SMP e NUMA
  3. Afinidade?
  4. Hints da Aplicação?
  5. Colocar CPUs ofline?
- 2.5 usa o scheduler de Ingo Molnar: http://kerneltrap.org/node.php?id=341

Outros Escalonamentos

- Fair-share, cada “share” tem uma percentagem do CPU e outros recursos (QoS):
  - `http://www.bell-labs.com/project/eclipse/`.
- Deadline Driven com vários tipos de deadline:
  - Hard são garantidas;
  - Soft tem probabilidade quantificada;
  - Time-Sharing e batch.
- 3-niveis com isocronico, tempo-real e time-sharing:
  - reserva de recursos: CPU, MEM, HD;
  - processos de tempo-real podem ser interrompidas por tarefas isocronicas em pontos bem-definidos (fecho de unidade de trabalho)
  - tarefas isocronicas usam escalonamento “rate-monotonic”.
  - Tarefas time-sharing são fully preemptible.
  - Evita *receive livelock* onde sistema só processa ints colocando rede como TR.

IPC em Unix

IPC permite:

- transferência de dados;
- partilha de dados;
- notificação de eventos;
- partilha de recursos especializada;
- controle de processos: debugger quer tomar conta dos eventos de outro processo.

IPC em Unix Original: Sinais

Sinais vêm desde Unix original:

- mecanismo assíncrono de notificação;
- muitos com significado prédefinido (`SIGUSR1` e `SIGUSR2`);
- `kill()` envia sinais para outro processo.
- `sigpause()` espera por sinal.
- `sigaction()` define handler.
- São caros porque emissor tem que fazer syscall e kernel tem que mexer na pilha do receptor.
- Banda limitada: apenas 31 em SVR4 e 4.3BSD, 64 em AIX, Linux.
- Úteis para notificação.

Pipes

FIFO não-estruturado e unidirecional:

- Escrever no fim e ler do princípio;
- Escritores bloqueam se pipe cheio e leitores se vazio.
- `pipe()` retorna dois file-descriptores, um para ler e outro para escrever. Descriptores podem ser passados entre processos.
- Usados pela shell, tem algumas limitações:
  - não suportam broadcast;
  - não conhecem limites de mensagem;
  - não se pode especificar o leitor

Mais Pipes

Implementação varia:

- Tradicionalmente: inode e entrada na tabela de ficheiros com pipe.
- BSD usa sockets, SVR4 usa streams, Linux usa código especializado com semáforo e `kmalloc()` (vd. `fs/pipe.c`).

Named pipes: `mknod` é usado para criar o pipe, que depois é acessível a processos.

- Vantagens: persistência, acesso para qq processo.
- Desvantagens: têm que ser removidos, não são tão seguros, consomem mais recursos, mais complicados de criar.
- Linux usa o mesmo código, e pipes pertencem a um `pipefs` onde é montado o arquivo.

Controle de Processos

`ptrace(cmd, pid, addr, data).`

- Permite a um processo:
  - ler ou escrever no espaço de um filho (incluindo área-u);
  - mexer nos registos;
  - criar *watchpoints* no espaço de endereçamento;
  - interceptar sinais;
  - criar ou alterar *watchpoints*;
  - continuar a execução de um filho parado;
  - andar passo a passo;
  - matar o filho;
- `cmd == 0` é usado pelo filho para indicar que está controlada por `ptrace()`, alterando comportamento para sinais e para `fork()`

`ptrace` em acção

- Parente usa `wait()` para esperar eventos que mudam o estado do filho.
- Filho envia `SIGCHLD` quando acontece alguma coisa.
- `exec` no filho resulta em `SIGTRAP` que pode ser controlada pelo pai.
- Quando `SIGCHLD` chega pai usa `ptrace()` para controlar.

Limitações:

- só pode controlar filhos imediatos;
- não permite apanhar processos em andamento;
- extremamente ineficiente;
- problemas com programs `setuid()`.

Sistemas modernos usam `/proc`: `fs/proc`, `kernel/ptrace.c` e `arch/i386/kernel/ptrace.c`.

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.
Ver `ipc` em Linux.

Semáforos em SYSV

- `semid = semget(key, count, flag)` array 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

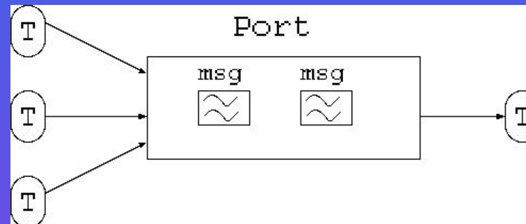
Troca de Mensagens é o mecanismo fundamental de comunicação:

- 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.

Bastantes melhoramentos em Mach 3.0.

Portas em Mach

Tasks tem direitos sobre portas de *send* e de *receive* (apenas a dono): comunicação muitos-para-um.

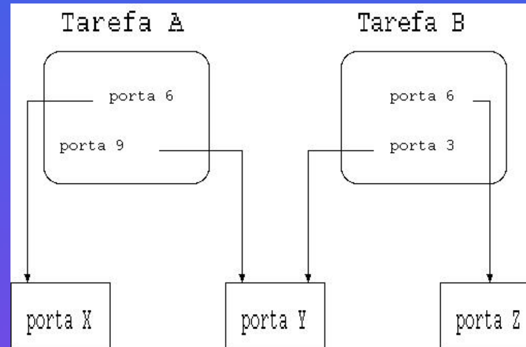


Mensagens podem ser:

- Simples: dados ordinários que não são interpretados pelo kernel;
- Complexa:
  - ⋆ dados ordinários, +
  - ⋆ memória *out-of-line* que é passada por referência (COW), +,
  - ⋆ direitos de envio ou recepção para portas.

  Kernel interpreta mensagens complexas.
- Cada porta tem um contador de referências.

Mais Portas



- Cada direito ou capacidade, é um nome para a porta. Nomes são inteiros e *locais* a tasks.
- Objectos do kernel são 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.

Portas, Tasks, e Threads

- Por Task:
  - ⋆ Cada task tem uma porta *task_self* para ela própria;
  - ⋆ Pode enviar para *bootstrap* que fornece acesso ao name server.
  - ⋆ uma porta de *exception*.
- Por Thread:
  - ⋆ direitos de envio para *self*;
  - ⋆ direitos de recepção para *reply*;
  - ⋆ uma porta de *exception*.
- Todos as threads numa task partilham direitos.

Mensagens em Mach

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

- Cabeçalho contém:

  Tipo: simples ou complexo;

  Tamanho: mensagegem inc. cabeçalho;

  Destino: uma porta;

  Resposta: uma porta, se necessário;

  ID: ao cuidado do usuário.

- Components contêm dados e descriptor:

  nome: tipo de dados, eg, memória interna, direitos de envio ou recepção, escalar (*byte*, *string*, *int de 16/32 bits*, . . .).

  Tamanho: tamanho de cada item de dados;

  Número: de items;

  Flags: dados são “in-line” ou “out-of-line” e se a memória ou os direitos devem ser deadlocados.

Estrutura de uma Mensagem

<table>
<tr><td colspan="2">tipo</td></tr>
<tr><td colspan="2">tamanho</td></tr>
<tr><td colspan="2">porta local</td></tr>
<tr><td colspan="2">porta destino</td></tr>
<tr><td colspan="2">ID</td></tr>
<tr><td>nome</td><td>tamanho</td></tr>
<tr><td>número</td><td>flags</td></tr>
<tr><td colspan="2">dados</td></tr>
<tr><td>nome
número</td><td>tamanho
flags</td></tr>
<tr><td colspan="2">dados</td></tr>
</table>

Interface

Três funções:

- ⋆ `msg_send()` envia sem esperar;
  ⋆ `msg_rcv()` espera por mensagens;
  ⋆ `msg_rpc()` envia e espera por uma resposta que pode vir no próprio buffer.
    ∗ Optimização de `msg_send()` + `msg_rcv()`.
    ∗ Originalmente o header tem o tamanho máximo da msg que pode *receber*.
    ∗ No fim o header tem o tamanho da mensagem.
- Todas operações têm `TIMEOUT`.

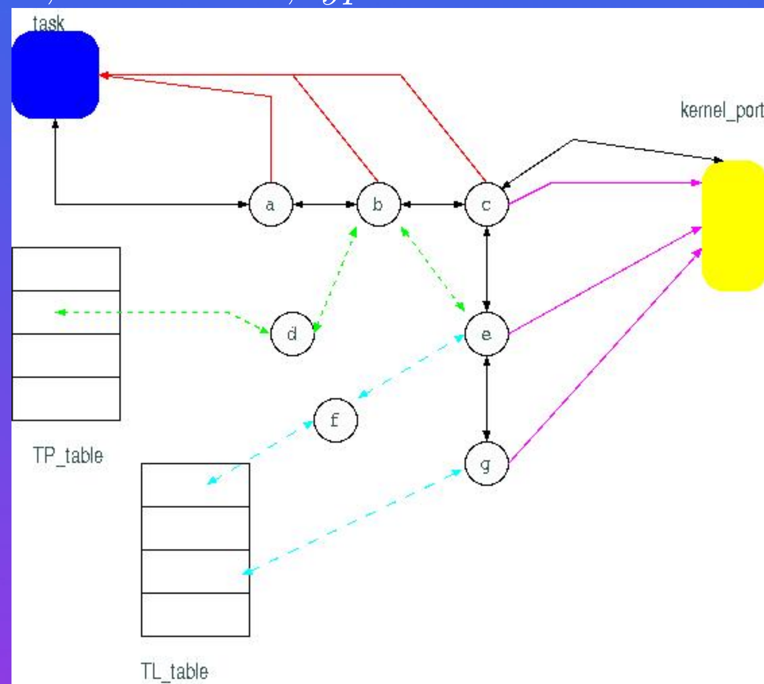
Implementação de Portas em Mach

Cada porta é uma fila protegida de mensagens no kernel:

- contador de referências para a porta;
- ptr para a task que tem direitos de recepcão;
- nome local no receptor;
- ptr para porta backup;
- lista dupl. ligada 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 (“backlog”).

Traduções de Portas

Tradução é $< task, port, local_name, type >$:



- sender usa $< task, local_name > (TL)$;
- receiver usa $< task, port > (TP)$;
- tasks têm que encontrar todos os direitos para a porta quando ela é dealocada;
- direitos são limpos quando a porta é destruída.

Passagem de Mensagens em Mach

Emissão:

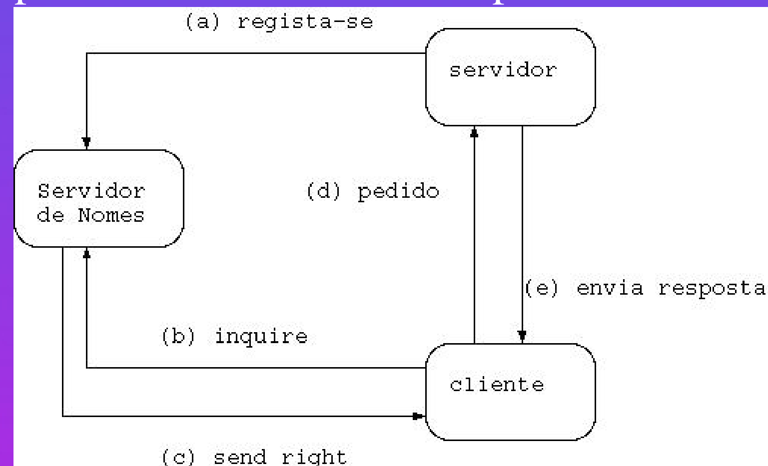
1. Emissor cria mensagem;
2. chama `msg_send()` do kernel;
3. kernel copia mensagem e:
   (a) se thread está à espera é acordado e recebe;
   (b) se lista cheia emissor bloqueia;
   (c) senão mensagem colocada na fila;

Recepção:

1. receptor chama `msg_rcv()`;
2. kernel chama `msg_dequeue()`;
3. kernel copia para receptor.

Portas em Mensagens

- Se emissor espera resposta, envia direitos para porta usando campo *reply port* na mensagem.
- Quando nova porta chega a task kernel traduz:
  1. se porta já existe *ok*;
  2. aloca novo índice ($<$ int) e cria nova tradução.
- *Servidor de nomes* passa direitos de acesso a portas de servidores:



Memória Out-of-Line

Mach usa copy-on-write:

- `msg_send()` chama `msg_copyin()`:
  1. modifica mapeamentos das pág para ser RO e COW;
  2. cria mapa temp. no kernel.
- `msg_rcv()` chama `msg_copyout()`:
  1. aloca espaço no receptor;
  2. copia entradas do mapa temp;
  3. remove mapa.
- Quando alguém tenta mexer na página, `PF` e kernel chama “fault handler”:
  1. cria uma nova cópia da página;
  2. muda mapa do processo que falhou;
  3. se puder, permite ao outro processo escrever na página original.
- Se emissor usar deallocate, `msg_copyin()` e `msg_copyout()` apenas passam as páginas.

Controle de Portas

- Mensagens podem ser enviadas no caminho lento (colocar na fila) ou caminho rápido (handoff scheduling).
- *backlog* é o limite configurável de mensagens numa porta.
- Notificações: mensagens enviadas para informar uma task de eventos:
  - `NOTIFY_PORT_DESTROYED`: qdo porta destruída msg. é enviada para porta backup;
  - `NOTIFY_PORT_DELETED`: qdo porta destruída msg. é enviada para todos os processos com direito de envio.
  - `NOTIFY_MSG_ACCEPTED`: se `msg_send()` usar `SEND_NOTIFY`, msg. é colocada mesmo que fila cheia e qdo msg. retirada da fila emissor kernel envia-lhe `NOTIFY_MSG_ACCEPTED`.

Operações sobre Portas em Mach

Destruição de portas: mensagens são removidas e NOTIFY_PORT_DELETED é enviado. Se mensagem contém direitos sobre a própria porta dá confusão em Mach 2.5.

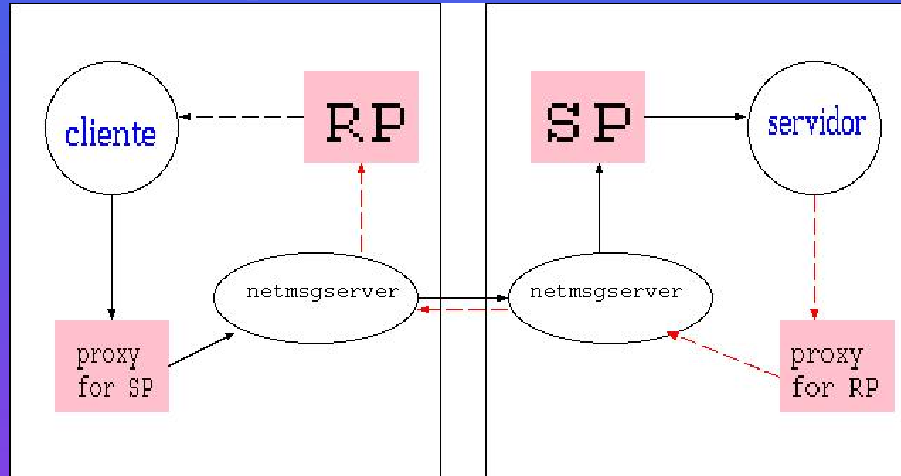
Portas backup: usadas quando a porta original é destruída.

Conjuntos de Portas: um receptor recebe todas as mensagens para o conjunto. Permite controle de vários objectos por uma única task.

Interpolação de portas: permite substituir uma capacidade para uma única porta com uma porta diferente. Usada por debugger para 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.
- MTBF: importante para "fault tolerant systems".

Paralelização de Unix exige muitas alterações:

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

Unix Tradicional

Reentrante e Non-preemptive:

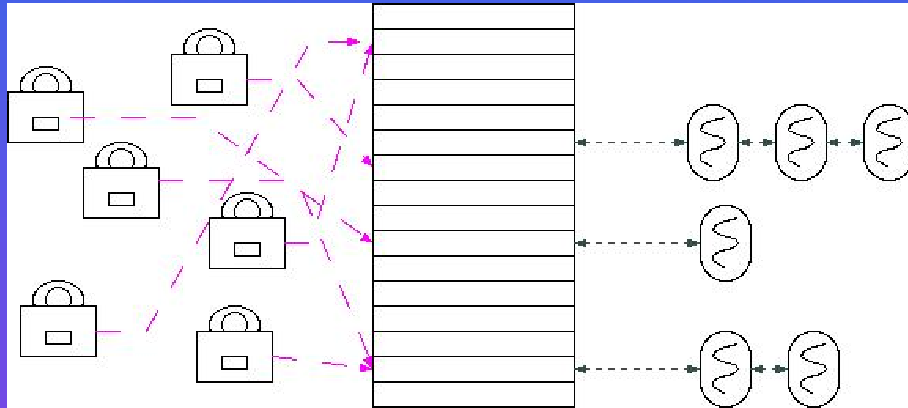
- Reentrante significa que vários processos podem estar no kernel;
- "Non-preemptive" significa que um processo não pode ser retirado do kernel.
- Interrupts controlados por `ipl`:
  - ⋆ Sistema só aceita interrupts com `ipl` superior;
  - ⋆ Linux apenas `irqsave` e `irqrestore` (vd. `include/asm-i386/system.h`:
    - ∗ em *x86* `cli` e `sti` são usados para desabilitar interrupções;
    - ∗ `pushfl` e `popfl` são usados para guardar o contexto corrente.

Recursos Partilhados

Recursos partilhados são controlados por flags `locked` e `wanted`:

- Quando uma thread precisa de um recuso partilhado (buffer de bloco), se `locked` limpo, coloca a 1 e entra;
- Se `locked` a 1, coloca `wanted` a 1 e bloqueia;
- quando o thread termina, limpa `locked` e verifica `wanted`: se a 1 percorre a *sleep queue* e acorda todos os threads;
- *acordar* é remover da fila, mudar estado para *runnable*, e colocar processo na fila do escalonador;
- processo depois recomeça do princípio

Sleep Queue



Recursos são mapeados numa *sleep queue*:

- recursos são associados ao *sleep channel*, habitualmente o endereço do recurso;
- função de hash mapeia o recurso para entrada na fila;
- Acorda-se todos os threads bloqueados no mesmo canal.
- Colisões.

Soluções: fila por recurso e *turnstiles*

Suporte a Multiprocessamento

A 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.
- x86 tem o prefixo `lock`, `xchgb` que faz swap atómico;
- Ultra-sparc tem swap condicional `casa`;
- Arquitecturas modernas precisam de `sync` ou `membar`.

Kernels para Multiprocessadores

Três variantes:

- Master-Slave: mestre pode ser único a realizar I/0 e a receber interrupts.
  - Facilita porting.
- Assimétricos funcionalmente: processadores especializados.
  - Exemplo: servidor de ficheiros Auspex NS5000.
- Totalmente Simétricos:
  - Memória Partilhada;
  - DSM;
  - Clusters.

Problemas com Sincronização em MPs

Mecanismo de Unix não funciona:

- Vários threads podem aceder a `locked` simultaneamente.
- Bloqueamento de Interrupts não funciona.
- *Wakeup perdido* um processo está a adormecer enquanto outro processo está a devolver 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 nunca conseguir chegar ao recurso.

Semáforos

Usados nas primeiras implementações de Unix SMP (IBM/370 e AT&T 3B20A):

- `P()` (down):

```
void P(semaphore *sem)
{
  *sem -= 1;
  if (*sem < 0) sleep();
}
```

- `V()` (up):

```
void V(semaphore *sem)
{
  *sem += 1;
  if (*sem <= 0) wakeup_a_thread();
}
```

- `CP()`: versão não bloqueante de `P`.
- Em Linux ver `include/asm-i386/semaphore.c` e `arch/i386/kernel/semaphore.c`.

Aplicações de semáforos

- Mutex: inicialmente a 1:

```
initsem(&sem, 1);
....
P(&sem);
usar recurso;
V(&sem);
```

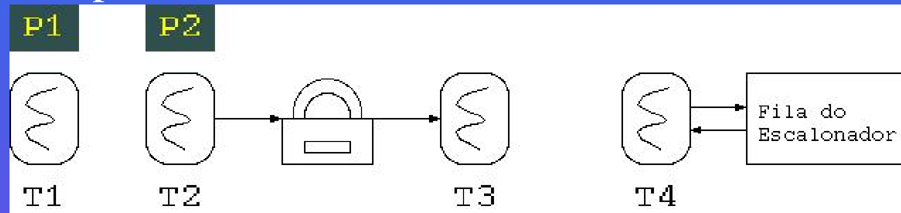
- Espera de eventos: inicializa a 0.
- Recursos numeráveis: inicialmente o número de recursos disponíveis.
- Linux: por `bdflush()` daemon, quota, directórios, `lookup()`, acessos a nós-i, proteger gestão de memória, alocação interna de memória, ...

Problemas com Semáforos

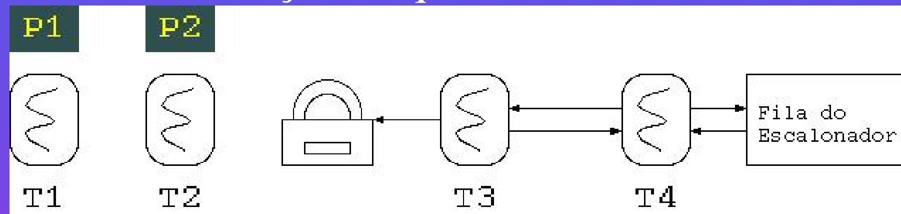
- Semáforos não têm spinning;
- Bloquear em semáforos pode ser lento porque exige manipulação de filas e mudança de contexto;
- Semáforos não dão garantia sobre o que estão a proteger:
  - `getblk()` encontra um bloco na cache;
  - faz `P()` no buffer e bloqueia;
  - não sabemos porque `getblk()` fez `P()` no semáforo;
  - mas temos que garantir que quando processo acordar está lá o mesmo bloco!

Comboios em Semáforos

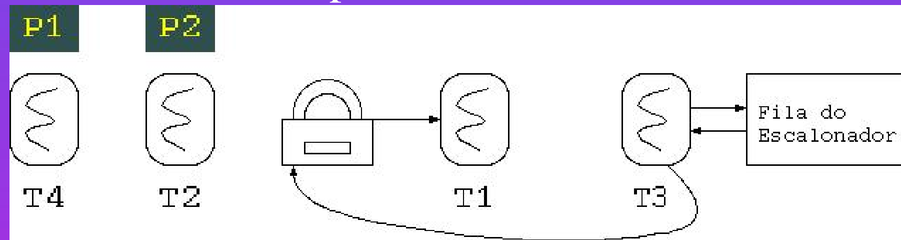
Comboios: problema típico de semáforos.



- Acontecem quando há contenção frequente:



- O thread que recebe o semáforo pode não estar executando:



- O thread que roda pode suspender no semáforo.
- Threads podem bloquear desnecessáriamente.

Spin-Locks

Mutexes com busy-wait. Para operações rápidas:

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

Evitar tráfego desnecessário (máquinas antigas):

```
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.
- Linux: `include/asm-i386/spinlock.c`.

Variáveis de Condição

Associadas a um predicado baseado em dados partilhados.

- `wait()`: espera pelo recurso

```
void wait(condition_t *c, spinlock_t *s) {
  spin_lock(&c->listLock);
  add self to linked list;
  spin_unlock(&c->listLock);
  spin_unlock(s);
  schedule();
  /* event has occurred */
  spin_lock(s);
}
```

- `signal()` e `broadcast()`: acorda um ou todos os processos.

```
void do_signal(condition_t *c) {
  spin_lock(&c->listLock);
  remove thread from linked list;
  spin_unlock(&c->listLock);
  if thread was removed, make it runnable.
}
```

- Usado em UTS.

Variáveis de Condição

- Semelhantes a channel: para evitar wakeups perdidos é necessário proteger com mutex.
- Obriga a um processo manter vários locks: não há problema se os locks forem ordenados.
- *Eventos*:
  1. awaitDone();
  2. setDone();

  Implementado por booleano `done`, spin-lock e variável de condição.
- Blocking locks podem ser implementados com variáveis de condição.

Locks de Leitura Escrita

Usar rwlocks:

- lockShared():

```
void lockShared(rwlock_p r) {
  spin_lock(&r->sl);
  r->nPendingReads++;
  /* don't starve writers */
  if (r->nPendingWrites > 0)
    wait(&r->canRead,&r->sl);
  /* exclusive at work ? */
  while(r->nActive < 0)
    wait(&r->canRead,&r->sl);
  r->nActive++; r->nPendingReads--;
  spin_unlock(&r->sl);
}
```

- unlockShared():

```
void unlockShared(rwlock_p r) {
  spin_lock(&r->sl);
  r->nActive--;
  if (r->nActive == 0) {
    spin_unlock(&r->sl);
    do_signal(&r->canWrite);
  } else spin_unlock(&r->sl);
}
```

Locks de Leitura Escrita: Exclusivo

- lockExclusive():

```
void lockExclusive(rwlock_p r) {
  spin_lock(&r->sl);
  r->nPendingWrites++;
  while(r->nActive)
    wait(&r->canWrite,&r->sl);
  r->nPendingReads--; r->nActive = -1;
  spin_unlock(&r->sl);
}
```

- unlockExclusive():

```
void unlockExclusive(rwlock_p r) {
  boolean_t wakeReaders;
  spin_lock(&r->sl);
  r->nActive = 0;
  wakeReaders = (r->nPendingReads != 0);
  spin_unlock(&r->sl);
  if (wakeReaders)
    do_broadcast(&r->canRead);
  else
    do_signal(&r->canWrite);
}
```

RWLocks: Comentários

- Outras: `tryLock()`, `upgrade()` e `downgrade()`.
- O que fazer quando se liberta um lock?
  - ⋆ Último 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.
- `upgrade()` corre o risco de deadlock.
- Linux: ver `include/asm-i386/rwlock.h`.
- Usados em código de rede, file-system, ...

Considerações

- *Contador de Referências*: necessários para quando se partilham objectos.
- *Prevenção de Deadlock*: locking hierárquico e estocástico,
  - ⋆ Geralmente, primeiro buffer e depois lista de blocos em disco;
  - ⋆ E se quisermos libertar um bloco da lista?
  - ⋆ Solução: `try_lock()`.
- *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.

Outras Implementações

- SVR4.2 MP suporta mutexes (com IPL), RW locks, sleep locks e variáveis de sincronização.
- Digital Unix Locks Simples (spin-locks) e Complexos (abstracções):
  - ⋆ uso partilhado ou exclusivo; bloqueamento; recursão.
  - ⋆ Também suporta `sleep()` e `wakeup()` com variáveis de condição.
- NCR introduziu *Advisory Processor Locks*, com *hints* (dormir ou spin) que podem ser voluntários ou mandatórios.
- Solaris usa locks adaptativos e turnstiles e fornece semáforos, RW locks e variáveis de condição.

Ficheiros em Unix

Noções fundamentais:

- *Arquivo* contém dados;
- *Sistema de Arquivos* permite organização desses arquivos;
- Interface:
  - ⋆ Chamadas de sistemas e utilitários que permitem manipulação de ficheiros.
- A interface tem sido estável, mas a implementação evoluiu muito:
  1. Múltiplos sistemas de arquivos (*S5FS*, *UFS*, *EXT2FS*, *FAT*, *LOGFS*,...);
  2. Sistemas distribuídos (*NFS*,*AFS*,*CODA*,*SMBFS*,...)

Organização de arquivos

Arquivos contém dados:

- Arquivos são uma sequência ordenada de bytes. Estrutura é problema da aplicação.
- Ficheiros são organizados hierárquicamente com *directórios* sendo os nós da árvore.
- Nome de ficheiros podem conter qualquer caracter excepto “/” e o caracter nulo.
- Processo tem *cwd*.
- *Pathname* indica como aceder a um arquivo.
  1. *caminhos absolutos* são desde a raíz;
  2. *caminhos relativos* são desde o *cwd*.

Directório

- Entrada num directorio é um *hard link*. Todos os hard links são equivalentes.
- Sistemas diferentes têm estutura de directório diferente.
- Unix moderno fornece estrutura `dirent` e conjunto de operações:
  - `opendir()` abre directório;
  - `readdir()` lê elemento;
  - `rewinddir()` volta ao princípio;
  - `closedir()` fecha directório.

Atributos de Ficheiros

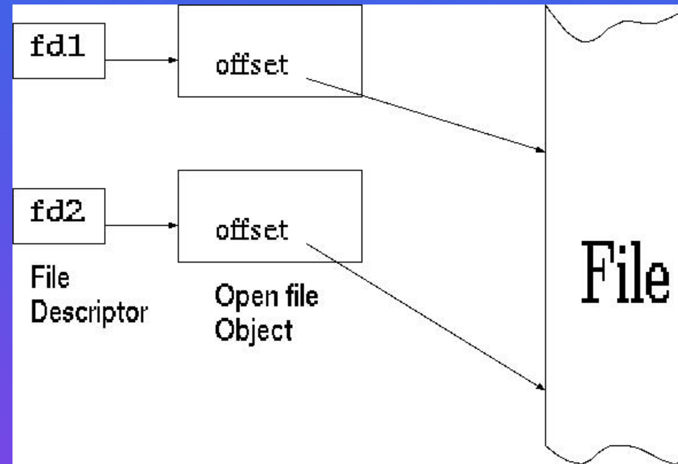
- *Tipo*: normal, directório ou especial.
- Número de *hard links*.
- Tamanho.
- DeviceID e Nó-i
- UID e GID do dono
- *Timestamps*: último acesso, última modificação, últ. mod. dos atributos.
- *Permissões*: leitura, escrita, acesso. Alt: ACLs.
- *Flags*: `suid`, `sgid`, `sticky`. `sgid` é usada para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

Descriptores de ficheiros

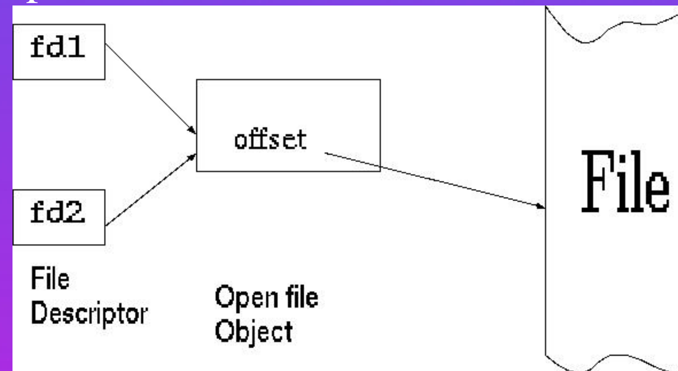
- Para ler arquivo é preciso abri-lo:

  `fd = open(path, oflag, mode)`
- `creat()` tb abre arquivo: `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;
  - por usuários diferentes ou o mesmo usuário.
- FD:
  - Representa a sessão cujo contexto está guardado no *open file object*.
  - Contexto inclui modo de aberta e *offset*.
  - Vários fds podem existir para o mesmo objecto:

Descriptores vs. Arquivos



- dup( ) e dup2( ) duplicam fd.



Descriptores vs. Arquivos

- Processos podem passar fd para outros processos (referência a objecto):
  1. SVR4 usa streams,
  2. BSD usa sockets com `sendmsg()`.

  Usado por *connection servers*.

I/O em Unix

Acesso pode ser sequencial ou “random”:

- Kernel mantém um offset, inicialmente 0.
- `lseek()` permite saltos (acesso “random”).
- `read()` e `write()` são semelhantes:

  ```
  nread = read(fd, buf, count);
  ```

  lê no máximo `count` caracteres e copia-os para `buf`.
- Operações de I/O são atómicas entre elas.
- `O_APPEND` permite abrir em mode append.
- Solaris fornece `pread()` e `pwrite()` (Linux tb).

Scatter-Gather

- `readv()` e `writev()` implementam *scatter-gather I/O*:
  - ⋆ I/O sobre vector de buffers;
  - ⋆ Útil por ex. para construir/ler pacotes;
  - ⋆ Diminui número de syscalls.

Locking

- Originalmente não suportado em Unix
- 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 adiciona locking mandatório, via `chmod()`;
  - ⋆ SVR4 adicona BSD através de `fcntl()` (`F_GETLK`, `F_SETLK` e `F_SETLKW` e de `lockf()`.

Sistemas de Arquivos

- Sistemas de ficheiros têm uma hierarquia. `mount()` coloca um sistema de ficheiros sobre outro e esconde o que estava antes.
  - ⋆ O que existia antes desaparece.
- Disco lógico abstrai armazenamento:
  - ⋆ disco
  - ⋆ partição
  - ⋆ Espelhos de discos (cópias de dados)
  - ⋆ Striping: distribui por vários discos
  - ⋆ RAID: combina espelhos e striping
  - ⋆ Volumes: trabalhar com vários discos como se um único.
- `newfs` ou `mkfs` constroem um novo disco;

Ficheiros Especiais

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

- Arquivos Normais.
- 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.
  - *Pathname* pode ser absoluto ou relativo.
- pipes e FIFOs:
  - geralmente bloqueiam;
  - `fnctl(fd,O_NDELAY)` faz com que não bloqueiem.
- devices que podem ser:
  - caracteres;
  - blocos

Múltiplos Sistemas de Ficheiros

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

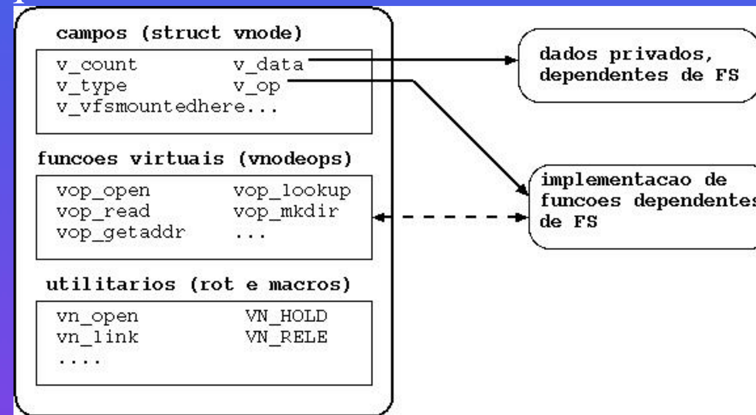
- Fazer como com devices I/O:
  - ⋆ Usar as operações gerais enquanto se puder;
  - ⋆ Quando é necessário fazer algo de específico existe uma array de operações:

```
struct cdevsw {
 int (*d_open)();
 int (*d_close)();
 int (*d_read)();
 int (*d_write)();
 ...
} cdevsw[];
```

  - ⋆ Cada dispositivo:
    1. fornece a sua rotina; ou,
    2. usa rotina geral (*default*).
- Programação por objectos!

Nó-v em Solaris

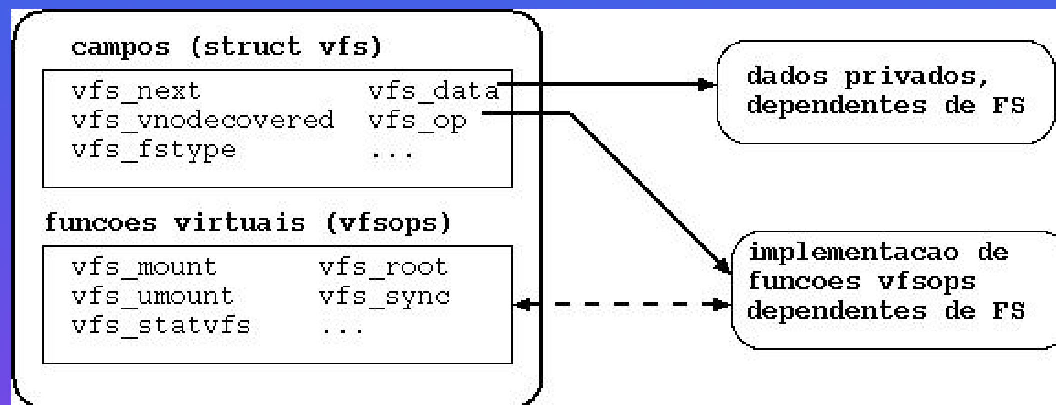
Nó-v representa um arquivo aberto:



- Cada *nó-v* 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.
- Macros simplificam acesso:

```
#define VOP_CLOSE(vp,...) \
  (*((vp)->v_op->vop_close))(vp,...)
```

Vfs em Solaris



Vfs representa um sistema de arquivos aberto:

- Abstracção;
- Operações diferentes: `mount()`, `umount()`;
- Conceitos básicos:
  - Mount point;
  - Lista de sistemas de arquivos montados.

Implementação de Nós-V: objectivos

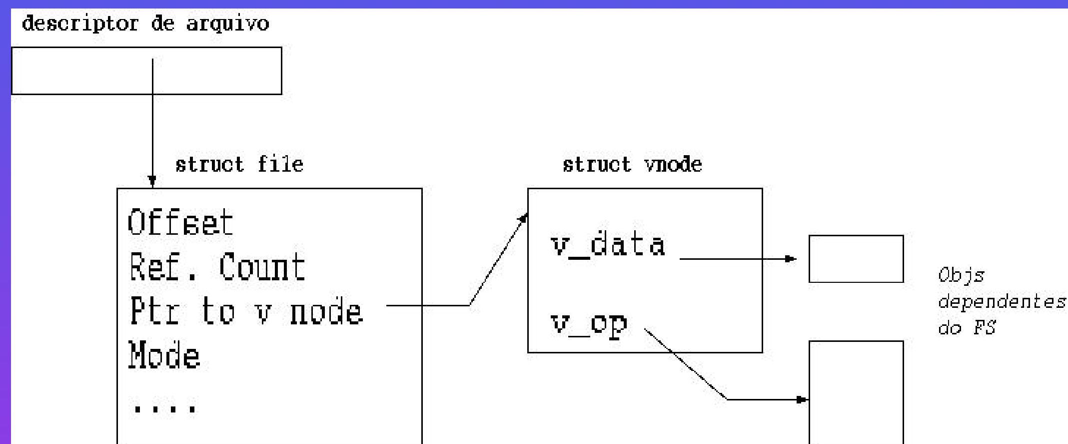
Ideia é poder usar a interface em sistemas de arquivos muito diferentes:

- Cada operação deve poder ser realizada em função do processo corrente, que pode adormecer se a função bloqueia.
- Locks para serialização devem ser libertados antes de operação completar.
- A interface deve ser *stateless* (`NFS`), evitando variáveis globais e campos na `u_area` para passar info. entre operações.
- a interface deve ser *reentrante*: substituir variáveis globais (`u_error` e `u_rval1`) por retorno de funções.
- Implementações devem poder usar recursos globais como a cache de buffers.
- Interface deve ser usável por um servidor.
- Evitar tabelas de tamanhos fixo.

Relação entre nós-v e ficheiros abertos

Processo pode aceder um nó-v ou via:

1. via *file descriptor*;
2. via *lookup* do nome.



Alocação de FD:

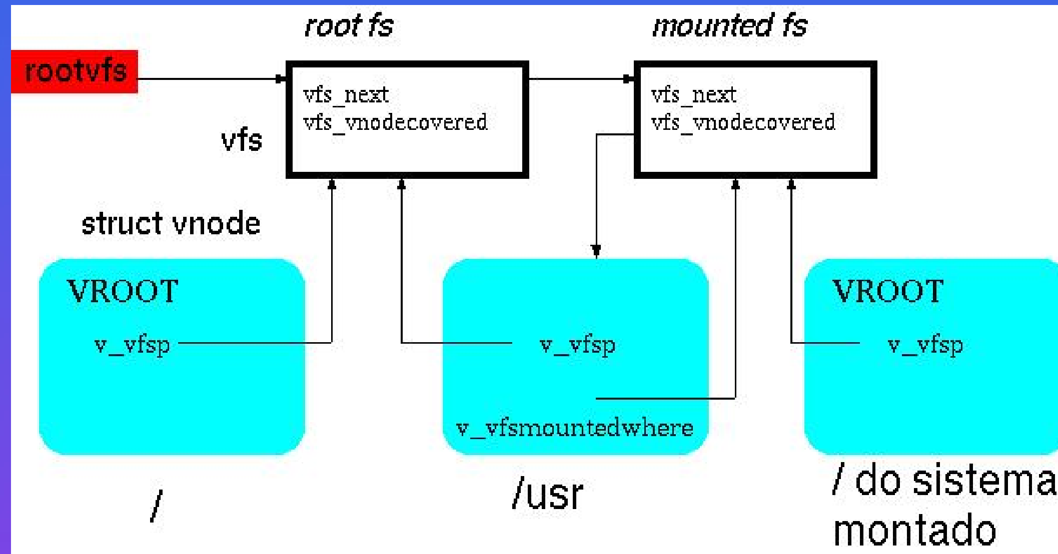
- Originalmente, array estático;
- Alocação dinâmicamente, eg, como uma lista ligada de blocos de 32 entradas.
- `kmem_realloc()`

Nó-V em Detalhe

Cada nó-v tem as seguintes estruturas:

- `v_flag`: raiz de FS, ...
- `v_count`: número de referências (ficheiros abertos, cdw, mount points, lookup). Importante para arquivos temporários;
- `v_fsnountedwhere`: para ponto de montagem;
- `v_op`: operações;
- `v_fsp`: file system;
- `v_stream`: stream associado;
- `v_page`: páginas residentes;
- `v_type`: tipo de arquivo;
- `v_rdev`: device ID;
- `v_data`: dados privados

Objecto 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;
  - ⋆ alocados juntamente com parte independente.
- Operações da interface no nó-v:

```
vop_open()     vop_close()
vop_read()     vop_write()     vop_ioctl()
vop_getattr()  vop_setattr()   vop_access()
vop_lookup()   vop_create()    vop_remove()
vop_link()     vop_rename()
vop_mkdir()    vop_rmdir()     vop_readdir()
vop_symlink()  vop_readlink()  vop_inactive()
vop_rwlock()   vop_rwunlock()  vop_realvp()
vop_getpage()  vop_putpage()
vop_map()      vop_poll()
```

- Operações da interface no vfs:

```
vfs_mount()  vfs_umount()
vfs_root()   vfs_statvfs()  vfs_sync()
```

Montagem de FS

SVR4 usa `vfssw[]`, um switch global com as características de cada FS:

1. `mount()` primeiro obtém o vnode do ponto de montagem com `lookuppn()`: nó-v tem que ser directório e nenhum outro FS pode estar montado nele;
2. Procura entrada em `vfssw[]`, dado tipo de FS;
3. Chama `vsw_init()`, específico ao FS;
4. Aloca novo `vfs`;
5. Inclui vfs na lista comandada por `rootvfs`.
6. `vfs_op` para `vfsops` de `vfssw[]`;
7. Instala `vfs_vnodecovered` para `nó-v` do mount point;
8. `vfs_vfsmountedwhere` do nó-v aponta para vfs;
9. chama `VFS_MOUNT()`;

VFS_MOUNT( )

Cada FS tem que implementar `VFS_MOUNT( )` à sua maneira:

1. Verificar permissões;
2. Alocar e inicializar objeto privado;
3. colocar um ptr. para ele em `vfs` $\longrightarrow$ `vfs_data`;
4. aceder ao directório raiz do FS e inicializar seu nó-v.

FS locais usam superbloco, FS distribuidos chamam o servidor.

Travessia do nome

`lookuppn()` recebe um nome e retorna um ptr para nó-v:

1. se não for último componente, usa `v_type` para saber se nó-v inicial é directório.
2. Se componente é `..` e cwd raíz, apanhe o componente seguinte.
3. se componente é `..` e cwd `VROOT`, acesse `v_vfsp` → `vfs_vnodecovered`.
4. Chame `VOP_LOOKUP()` no directório corrente: retorna ptr. para nó-v do arquivo e obtém um hold.
5. se componente não fôr encontrado:
   (a) se fôr último, retorne sucesso, passando ptr para o pai e mantendo hold;
   (b) senão `ENOENT`.
6. se `v_vfsmountedhere != NULL` encontre o vfs correspondente e chame `vfs_root()` para encontrar o *nó-v raiz*.

Travessia do nome

1. Se `v_type == VLNK`, traduza com `VOP_SYMLINK()`, junte a tradução e reinicialize (se caminho absoluto, comece da raíz):
   - arg. de `lookuppn()` pode suprimir avaliação de links simbólicos no último componente (`lstat`);
   - `MAXSYMLINKS` limita o número de links simbólicos numa travessia.
2. Liberta directório (segurado ou por `VOP_LOOKUP` ou por inicialização.
3. Volte ao principio e procure novo componente.
4. Se procurou todos, mantenha o hold e devolva um ptr para o nó-v.

Cache de Acesso a Directórios

- Cache *LRU* contendo nó-v de directório, nome de arquivo no directório, e ptr para nó-v do arquivo:
  - Organizado por $dir + nome$.
- `VOP_LOOKUP` procura lá primeiro:
  - se encontrar, incrementa ctr. de refs;
  - senão, procura dir. e adiciona entrada na cache.
- Arquivo pode ser removido e nó-v usado para outro arquivo:
  - Em SVR4 cache tem ref. para o nó-v;
  - não podemos libertar o nó-v, 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, ...,`

- `tvp` tem o resultado:
- Algoritmo
  - ⋆ 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()`

1. aloca um fd.
2. aloca um objecto ficheiro.
3. chama `lookuppn()` 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 `ENOENT`.
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 índice do fd.

Análise

SVR4 e Solaris:

- kernel locka o nó-v antes de fazer leitura ou escrita: garante sequencialidade;
- Atributos são implementados por estrutura `vattr`, baseada no s5fs.
- Credenciais são passadas por referência a um objecto de `u_area` ou `proc`.
- Vantagens: portátil, genérico.
- Desvantagens:
  - ⋆ `lookuppn()` chama `VOP_LOOKUP` para cada componente, devido a NFS: Gera tráfego desnecessário.
  - ⋆ operação não faz locking do directório pai: `open()` tem que verificar se outro processo criou o arquivo que queremos criar, causando overheads em `VOP_CREAT()`.
  - ⋆ Dependências em gestão de memória, OS.

VFS em 4.4BSD

Optimizações:

- Usa modelo *stateful*;
- `namei()` chama `lookup()` que pode passar vários componentes, sem atravessar mount-point;
- argumentos de `namei()` estão em `nameidata`. Se razão é crear ou remover, obtém lock;
- `abortop()` desiste do lock;
- Protocolo é implementado na componente dependente: NFS não precisa de obter lock;
- Problema: serializa operações no directório;
- Cada processo mantém uma cache do dir. e offset do último name lookup.

VFS em OSF/1

- Objectivos:
  1. evitar operações redundates;
  2. manter *stateleness*;
  3. funcionar com SMP e UP.
- Informação de estado é passado com *hint*, associado a,
- *timestamps*, para verificar se o directório não foi alterado.
- mutex protege metadados de arquivos.

VFS em Linux

- *dentry* corresponde a entrada no directório de cache;
- inode corresponde a `nó-v`;
- vfs tem uma noção de superbloco:

```
struct file_system_type {
  const char *name;
  int fs_flags;
  struct super_block *(*read_super)
   (struct super_block *, void *, int);
  struct module *owner;
  /* For kernel mount, if it's FS_SINGLE*/
  struct vfsmount *kern_mnt;
  struct file_system_type * next;
};
```

  ⋆ `name` eg, “ext2”;

  ⋆ `fs_flags`: `FS_REQUIRES_DEV`, `FS_NO_DCACHE`, . . .

  ⋆ `read_super`: método a chamar quando montamos nova instância;

  ⋆ `next`: lista de mounts.

Superblocos

- `read_super` recebe um superbloco e opções de `mount`.
- Estrutura `super_block` inclui:
  - info. sobre lista de sbs;
  - tamanho de blocos;
  - lock;
  - flag `dirty`;
  - lista de operações;
  - tipo;
  - quota;
  - pointer para *dentry* da raíz;
  - `wait_queue`;
  - device.
  - Union para info específica.

Operações

- `read_inode`: lê nó-i do FS;
- `write_inode`: escreve nó-i no FS;
- `put_inode`: chamado qdo nó-i é removido da cache;
- `delete_inode`: chamado para remover nó-i;
- `notify_change`: chamado qdo atributos do nó-i são alterados (senão `write_inode()`) (BGL);
- `put_super`: umount chama VFS que quer libertar superbloco (superblock lock);
- `write_super`: VFS precisa de escrever;
- `statfs`: obter estatísticas do FS (BGL);
- `remount_fs`: chamado por remount (BGL);
- `clear_inode`: chamado para libertar nó-i;
- `umount_begin`: chamado no princípio de `umount`.

inode_operations

```
create()      lookup()
link()        unlink()      symlink()
mkdir()       rmdir()       mknod()
rename()      readlink()
readpage()    writepage()   bmap()
truncate()    permission()  smap()
updatepage()  revalidate()
```

- Chamados sem locks;
- recebem dentries;
- `create()` só é preciso para arquivos regulares, recebe dentry não instanciado;
- `d_instantiate()` é usado para criar uma nova dentry;
- `lookup()` encontra nó-i em parente e chama `d_add()` para adicionar nó-i numa dentry;
- `lookup()` segura o semáforo do directório pai.

file_operations

```
llseek()      read()    write()
readdir()     poll()    ioctl()
mmap()        open()    release()
fsync()       fasync()  check_media_change()
revalidate()  lock()
```

- `poll()` é chamado por `select()` e `poll()`;
- `open()` cria um novo `struct file` e inicializa `f_op` com defaults;
- qdo `open()` abre device chama rotinas no kernel que substituem as rotinas do FS com as do device driver;
- `fasync()` é chamado para `fsync()` em modo assíncrono;

FS e dentries

Manipulação de dentries:

- `dget()` abre nova handle, incremente uso;
- `dput()` fecha uso: decrementa e se chegar a 0 `d_delete()`;
- `d_drop()`: remove dentry da lista de hash do seu pai;
- `d_delete()` remover dentry, se última ref. passa a ser negativa (`d_iput()`);
- `d_add()` junta à lista dos seus pais e chama `d_instantiate()`;
- `d_instantiate()` adiciona à lista de hash do nó-i, inicia (incrementa) `i_count`. Habitualmente, chamada qdo nó-i é criado.

dentry_operations

- `d_revalidate()`: é chamada para revalidar, habitualmente `NULL`;
- `d_hash()` adiciona dentry na tabela de hash;
- `d_compare()`: compara duas dentries;
- `d_delete()` qdo a última ref. é removida;
- `d_release()` qdo dentry for dealocada;
- `d_iput()` chamado qdo entry perder o seu nó-i.
- Comece por `include/linux/fs.h`;
- Depois veja `include/linux/dcache.h` e `fs/`

Partições s5fs

B	S	Lista de No-I	Blocos com Dados

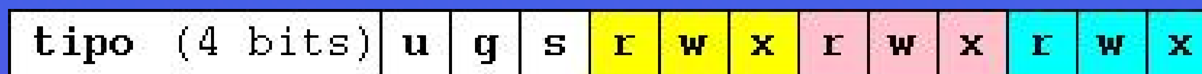
- Directório: lista linear de registos com 16 bytes:
  - Primeiros 2 bytes contêm nó-i
  - Os outros 14 contêm o nome do ficheiro.
  - As primeiras entradas são sempre `'.'` e `'..'`.
  - Entrada a 0 não existe.

Nós-I

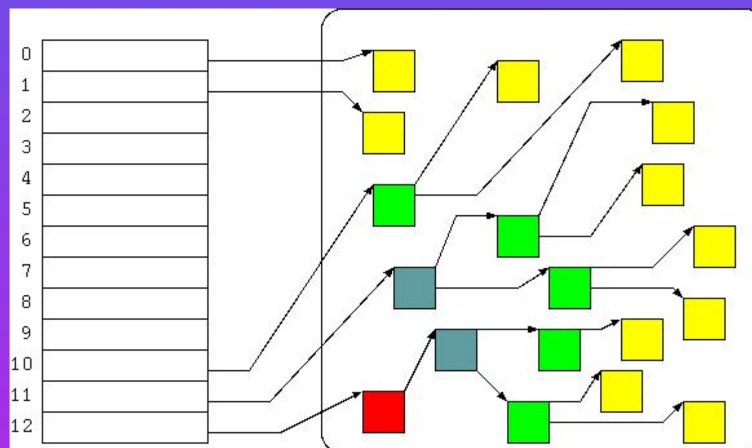
- nó-i:

Campo	Tamanho	Descrição
`di_mode`	2	tipo e permissões
`di_nlinks`	2	núm de hard links
`di_uid`	2	UID do dono
`di_gid`	2	GID do dono
`di_size`	4	tamanho (B)
`di_addr`	39	endereços de blocos
`di_gen`	1	núm de geração
`di_atime`	1	último acesso
`di_mtime`	1	última modificação
`di_ctime`	1	última mudança no nó-i

Campos do nó-i



- di_mode: suid, sgid, sticky, owner, group, others.
- Primeiros 10 campos são directos e existe um campo indirecto, outro duplamente indirecto, e outro triplamente indirecto:
- Podem existir buracos, o que causa problema para **tar** e **cpio**.



O Superbloco

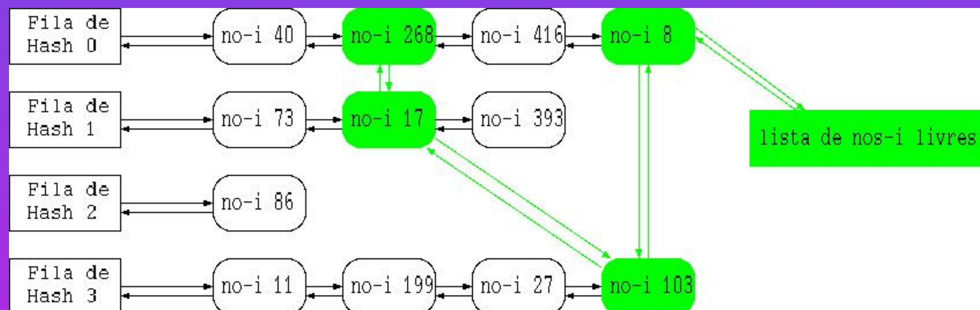
Inclui:

- Tamanho em blocos do sistema de ficheiros.
- Tamanho da lista de nós-i.
- Núm. de blocos e nós-i livres.
- 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.

Nós-i no Kernel

Nó-i em memória inclui:

- nó-v associado;
- Device ID da partição;
- Número do nó-i no FS;
- Flags para sincronização e gestão de cache;
- Ptr. para uma lista de nós-i livres;
- Ptr. para uma lista fila de hash;
- Núm. do último bloco lido.



Lookup

1. `lookuppn()` usa `VOP_LOOKUP()` para encontrar componente que chama `s5lookup()`.
2. `s5lookup()` primeiro procura na cache;
3. Senão, anda no directório;
4. se encontrar obtém o número do nó-i e chama `iget()` para o encontrar;
5. se nó-i na tabela de hash, tudo bem; senão
6. aloca nó-i e inicializa lendo do disco;
7. aloca e inicializa nó-v;
8. retorna um ptr. para nó para `lookuppn()`.

Apenas `iget()` aloca e inicializa nós-i.

File I/O: Leitura

Recebemos FD, user address e count:

- Código independente obtém `struct file` e verifica modos;
- Chama `VOP_RWLOCK()` para consistência;
- `VOP_READ()` chama `s5read()`;
- `s5read()` traduz offset para bloco e lê *uma página de cada vez*:
  1. mapeia bloco na VM do Kernel,
  2. chama `uiomove()` para copiar para espaço de usuário,
  3. `uiomove()` chama `copyout()` que gera PF se não está mapeada,
  4. Fault handler chama `VOP_GETPAGE()`.

File I/O: Leitura

- `s5getpage()` chama `bmap()`
  - ⋆ converter núm. de página
  - ⋆ procura nó-v para ver se pág. em memória
  - ⋆ senão, aloca pág. livre e chama disk driver.
- Quando I/O completa processo continua em `copyout()`, que deve verificar end. antes de copiar;
- `s5read()` regressa qdo dados tiverem sido copiados, ou em erro.

File I/O: Escrita

`write()` é semelhante mas:

- discos modificados continuam em memória;
- `write()` pode obrigar a alocar mais blocos;
- `write()` pode alterar bloco.

Alocação de Nós-I

- Quando o núm. de refs vai a 0 FS chama `VOP_INACTIVE()` que liberta o nó-i;
- Sistemas modernos mantêm o nó-i na lista livre, mas não o invalidam: `iget()` pode reusar;
- Tamanho da tabela de nós-i limita o número de nós-i activos: SVR4 usa LRU para limpar:
  - ⋆ nós-i podem ter páginas, não é boa ideia removê-los;
  - ⋆ Ideia: colocar o nó-i no fim da lista se *não* tiver páginas.
- Possível usar um alocador de memória para aumentar núm. de nós-i:
  - ⋆ se primeiro nó na lista livre ainda tem págs, coloca-o no fim da fila e aloca mais um nó-i.

Análise de s5fs

- A maior vantagem é simplicidade.
- Sistema depende muito do superbloco.
- Desempenho pode ser mau porque nós-i estão longe dos dados correspondentes.
- Alocação de blocos torna-se rápidamente aleatória.
- Tamanho de blocos demasiado rígido: 512B originalmente.
- Limites na funcionalidade: tamanho máximo do ficheiro.

Introdução a FFS

- Discos dividos em *grupos de cilindros*.
- Superbloco por grupo:
  - ⋆ 2 partes: informação sobre o FS e sobre o grupo;
  - ⋆ Colocados em offset variável do princípio do supergrupo.
- Blocos são divididos em blocos (8KB) e fragmentos (1KB ou 512B).
  - ⋆ Maior throughput;
  - ⋆ Dupla indirecção consegue 4GB;
  - ⋆ Apenas o último bloco pode ter fragmentos, o que pode obrigar a cópia:
    1. Se o último bloco ocupar um fragmento;
    2. Outros arquivos tiverem outros fragmentos;
    3. Temos que copiar para outro bloco se arquivo crescer.

Mais FFS

- Politica de alocação favorece grupos de cilindros:
  - Arquivos do mesmo dir. no mesmo grupo;
  - Novos directórios em grupos diferentes;
  - Dados no mesmo grupo que nó-i;
  - Mudar de grupo quando tamanho atinge 48KB e depois 1MB;
  - Tentar optimização alocação de blocos consecutivos.
- Extensões: nomes longos e links simbólicos.
  1. Nó-i;
  2. tamanho de alocação;
  3. tamanho do nome;
  4. nome.

FFS: Análise

- Ganhos substanciais em desempenho (VAX/750):
  - read throughpout de 29KB/s para 221 KB/s;
  - write througput de 48KB/s para 141KB/s.
- Fragmentos evitam overhead de espaço;
- Menos blocos indirectos;
- Necessário espaço livre (FFS com 4KB/1KB perto de s5 com 1KB);
- Discos modernos (SCSI e IDE) fazem gestão de cilindros.
- Melhorias possíveis:
  1. Escritas múltiplas;
  2. Pré-alocação de blocos para arquivos que estão a crescer.

EXT2

ext2fs:

- Derivado de MINIXFS e de EXTFS;
- Estrutura semelhante a FFS;
- Usa bitmaps para nós-i e blocos livres;
- Suporta atributos para arquivos e directórios:
    1. secure deletion,
    2. immutable,
    3. append-only;
- Grupos de blocos não dependem de layout físico;
- Pré-aloca 8 blocos adjacentes quando aloca um bloco;
- Suporta readahead;

Buffer Cache

Originalmente usada para evitar acesso a disco:

- *Data buffers* guardam blocos;
- Cerca de 10% de memória total;
- *Backing Store*: disco;
- *Write-back:* evita problemas de performance;
- Informação sobre directórios e nós-i é *write-through*;
- Unificada com VM em sistemas recentes.

Funcionamento da Buffer Cache

- Processo procura em tabelas de hash baseadas no *device* e no *número de bloco*:
  - ⋆ miss: kernel aloca novo buffer e lê novos blocos do disco;
  - ⋆ write: kernel coloca `dirty`;
  - ⋆ Interrupt handler pode mexer no buffer: interrupts devem ser desactivados para mexer em bloco.
- Lista de blocos livres é *LRU*:
  - ⋆ buffers inválidos (arquivo removido, erro) podem ser colocados na frente da fila;
  - ⋆ dirty buffers que chegam na frente são colocados na write queue do driver e depois enviados para a frente.

Buffer Header

- Buffer Header:
  - ⋆ identifica e localiza buffer;
  - ⋆ sincroniza acesso (lock);
  - ⋆ gestão da cache;
  - ⋆ interface ao disco;
- Vantagens
  - ⋆ Reduz tráfeho (90%);
  - ⋆ Interface com o disk driver (eg, alinhamento)
- Problemas:
  - ⋆ *Write-back* tem problemas com falhas;
  - ⋆ Dados são copiados 2 vezes: problemas para acessos sequenciais a arquivos grandes devido a *cache wiping*;
  - ⋆ Vxfs permite fornecer *hints* sobre arquivos.

Consistência

Problema se houver crash:

- Writes de dados não são grande problema: não comprometem consistência;
- `sync()` força writes;
- `fsflush()` executa `sync()` de 30 em 30 sec (ver `fs/buffer.c`);
- Perda de metadados pode tornar o sistema inconsistente:
  - sincronismo: garantir ordem de escrita de metadados (nós-i antes de alteração de dir);
  - `fsck()`

Problemas com Sistemas Tradicionais

Desempenho: layout de FFS ou EXT2 não usa banda total do disco:

- Atraso rotacional,
- Escrever uma pista de cada vez,
- Predominância de writes (2 para 1),
- Seeks devido a time-sharing;

Recuperação de crash: não é garantida;

Segurança: Permissões Unix (ACL seria melhor?)

Tamanho: Arquivos têm que caber em partição, existem limites arbitrários.

Journalling

Escrever tudo num arquivo append-only, o *log*:

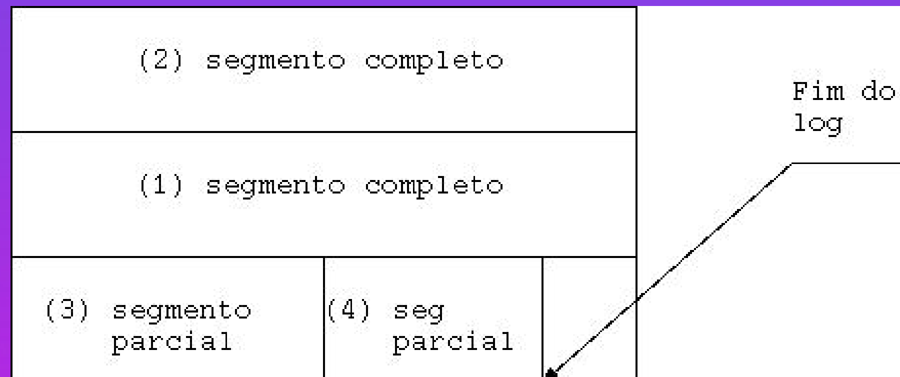
- O que colocar? Todas as modificações ou metadados?
- Colocar operações (alteração do bitmap) ou os resultados (blocos)?
- Suplementar (*log-enhanced*) ou substituir (*log-structured*)?
- *Redo-only* guarda apenas os dados modificados, simplifica recuperação mas coloca constraints em como colocar, *redo-undo* guarda valor novo e antigo, maior, recuperação mais complexa, mas $>$ concorrência. Qual o melhor?
- Garbage Collection para log-finito?
- Commit em grupo abre janela de vulnerabilidade?
- Como retirar dados do log em *log-structured*?

Log-Structured

- Única estrutura é o log;
- Writes são na cauda do log, são sequenciais, elimina seeks;
- Cada transferência são muitos dados, aproveita a banda;
- Recuperacão é muito rápida;
- Problema: buscar dados depemnde de cache;
- BSD LOG-FS baseado no trabalho de Sprite

Estrutura de BSD LOG-FS

- Mantém directório e nó-i;
- Nós-i são encontrados via *inode map*, que mapeia nó-i para endereço (guardado em disco, guardado no log em check-points);
- Alocação por segmentos (1/2MB).
- Segmentos parciais devido a falta de memória ou NFS:
  - CRCs;
  - Endereços dos nós-i;
  - Info para cada arquivo;
  - Flags.



LFS: Escrita

- Se controlador suporta *scatter-gather* discos são escritos directamente de cache;
- Endereços de blocos disco são atribuídos apenas na fase de escrita do bloco de disco;
- Bloco anterior é recuperável por *cleaner*;
- Cada write escreve *todos* os buffers dirty da cache: recuperação completa é possivel.
- read-only *ifile* guarda modificações só em *checkpoints*. Contém:
  - Mapa de nós-i;
  - Tabela de uso de segmentos: dados não obsolteos por segmentos, e tempo em que segmento foi alterado;
  - Recuperável dos segmentos.

LFS: Análise

- Problemas:
  - ⋆ Alterações num directório podem não entrar no mesmo segmento parcial;
  - ⋆ Alocação de blocos ocorre quando o segmento é escrito: cuidado com falta de espaço.
  - ⋆ requer muita memória física para cache.
- Vantagem em desempenho discutível:
  - ⋆ Performance melhor que Sun-FFS em operações que mexem em meta-dados;
  - ⋆ Sun-FFS tende ser melhor em I/O intensivo
- Logging de metadados pode trazer os mesmo benefícios com menor custos.

Logging de Meta-Dados

- Mantém estrutura normal;
- Log só é lido em caso de crash;
- Log pode ser arquivo ou externo;
- Pode afectar performance: *batching* combina várias alterações numa entrada.
- Wrapping do log?

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 expansivel incrementalmente.
- *Mobilidade de Ficheiros*: mudar a localização física.

Diferente de SO distribuído como Amoeba.

Considerações de Desenho

- Espaço de nomes:
  - ⋆ uniforme, todos os clientes usam o mesmo nome para o mesmo arquivo;
  - ⋆ ou, adaptável por cada cliente.
- Servidor com ou sem *estados*:
  - ⋆ offset no servidor para leitura;
  - ⋆ cada cliente envia offset de leitura;
  - ⋆ Não ter estados simplifica consistência e recuperação de falhas.

Considerações de Desenho

- Semânticas de Partilha:
  - ⋆ Unix exige que mudanças de um cliente sejam visiveis para os outros imediatamente;
  - ⋆ Semânticas de sessão são propagados no `open()` e no `close()`.
- 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:
  - Código de servidor separado de cliente;
  - Comunicação via RPC síncrona.
- Cada servidor exporta um ou mais sistemas, para leitura ou leitura-escrita.
- Montagem pode ser
  - *hard*, tenta até conseguir
  - *soft*, desiste após um bocado
  - *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).

Protocolo NFS

NFSv2, protocolo definindo operações e seus argumento:

Proc	**Entrada → Saída**
NULL	
GETATTR	fhandle → status, fattr
SETATTR	fhandle,sattr → status, fattr
LOOKUP	dirfh,name → status, fhandle, fattr
READLINK	fhandle → status,link_value
READ	fhandle,offset,count,totcount → status,fattr,data
WRITE	fhandle,offset,count,totcount,data → status,fattr
CREATE	dirfh,name,sattr → status,fhandle,fattr
REMOVE	dirfh,name → status
RENAME	dirfh,name1,dirfh,name2 → status
LINK	fhandle,dirfh,name → status
SYMLINK	fhandle,dirfh,name → status
MKDIR	dirfh,name,sattr → status,fhandle,fattr
RMDIR	dirfh,name → status
READDIR	fhandle,cookie,count → status,dir_entries
STATFS	fhandle → status,file_stats

Componentes de NFS

Outros componentes:

- **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
  - *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
  - ⋆ metadados:
    - ∗ directórios
    - ∗ inodes

O Conjunto de Protocolos: XDR

- XDR usa uma representação desenhada para o SUN:
  - inteiros
  - objectos opacos
  - strings
  - vectores
  - estruturas

O Conjunto de Protocolos: RPC

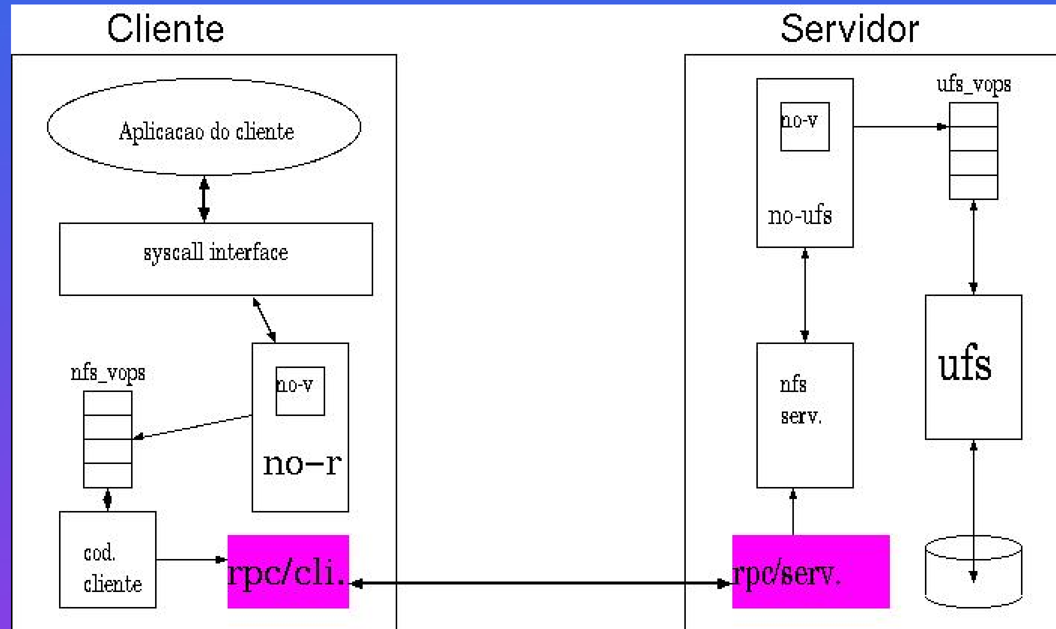
- Protocolo síncrono
- Fiável
  - apesar de habitualmente implementado sobre UDP
- Campos:
  - xid,
  - direcção
  - rpc_vers
  - programa e sua versão
  - informação
  - autentificação

Autorização

Diferentes tipos de protocolos de autorização:

- `NULL`
- `UNIX`
- `SHORT`, usada depois do primeiro pedido Unix
- `DES`
- `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 é lento.

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, o cliente controla as permissões.
- Ficheiros removidos continuam acessíveis em Unix.
  - Em NFS cliente muda a operação para um rename, e no fim um remove.
  - Problema: cliente pode crashar, arquivos temporários.
- Reads e writes podem involver várias operações e não são portanto atómicos entre clientes:
  - usar NLM.

Problemas de Performance de NFS

Problemas:

- Operações que alteram o servidor são extremamente lentas porque esperam que os writes sejam colocados no armazenamento.
- obter atributos obriga a um RPC por ficheiro: problema com `ls -l`.
- Retransmissões podem piorar situação em redes sobrecarregadas.

Melhorando Performance de NFS

- 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.
- Não atrasar cliente:
  - `write()` assíncrono para blocos completos;
  - atrasar `write()` de 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:
  - Aumenta latência;
  - Reduz número de escritas no servidor;
  - Útil se vários biods no cliente.

Cache de Retransmissões

Período de espera antes de retransmissão é muito curto (1-3sec):

- Pacotes perdidos podem acontecer com UDP.
- Operações *idempotentes*, como `READ()` podem ser executadas várias vezes, não há problema;
- *não idempotentes*, como `CREATE` ou `REMOVE` podem dar problemas.
- Solução: manter uma cache de pedidos que podiam dar problemas. Se pedido falha, verifica se é uma retransmissão e está na cache.
- Sun implementou solução incompleta: só verificava pedidos que falhassem;
- Digital faz cache de todos os pedidos e verifica sempre:
  - timestamps são usados para verificar se entrada é recente;
  - pedidos podem ser servidos de cache.

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:
  - ⋆ UID remapping;
  - ⋆ Root remapping.
- NFSv3:
  - ⋆ Writes asssíncronos como `COMMIT`:
    - ∗ `WRITE` é assíncrono;
    - ∗ dados só são retirados de cache quando cliente fecha arquivo;
    - ∗ aí cliente faz `COMNIT`
  - ⋆ Usa campos de 64bits;
  - ⋆ `READDIRPLUS` retorna info para todos os arquivos no directório;

Conceitos 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 alocação de memória.
- Permitir partilha

VM em Unix

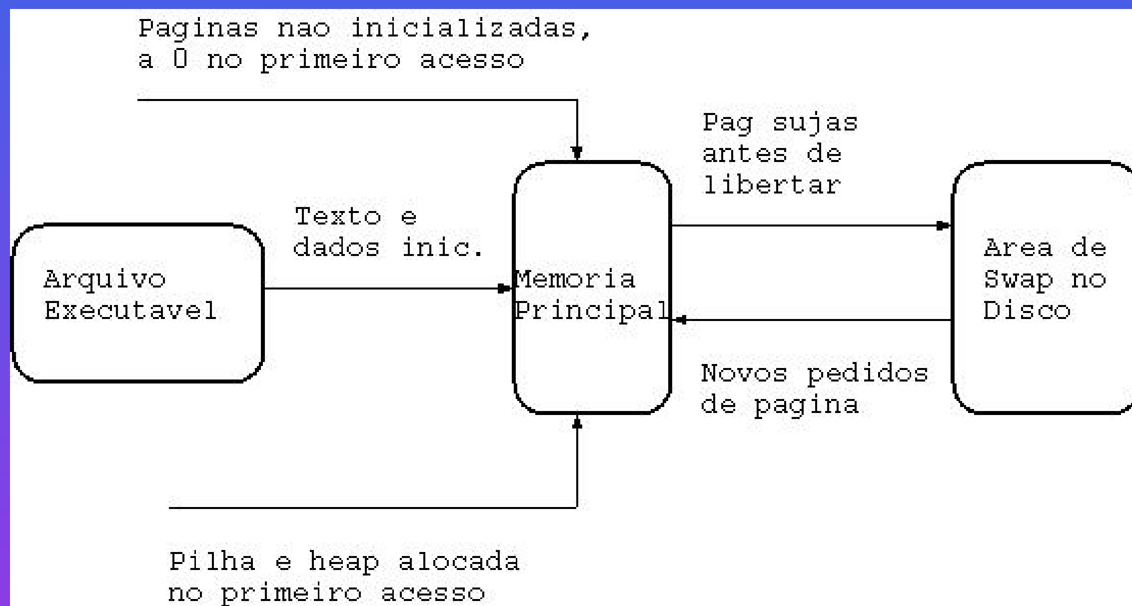
- Originalmente segmentos de 64KB(+64KB):
  - ⋆ programadores faziam “software overlay”;
- Swapping usado para libertar espaço;
- Paginação sobre demanda: 3BSD no VAX 11 (1978);
- Nova versão em SunOS 4 influenciou o desenvolvimento de paginação para SVR4 e Solaris;
- Paginador para Mach foi eventualmente adaptado para BSD4.4.

Requerimentos

Precisamos de:

- Gestão do espaço do endereços do processo;
- Tradução de endereços (via MMU e PF);
- Gestão optimizada da memória física;
- Proteção de memória:
  - ⋆ processos não podem ler páginas do kernel ou de outros processos;
- Partilha de Memória:
  - ⋆ partilha de parte do espaço de enderaçamento;
  - ⋆ partilha de frames (eg, depois de `fork()`).
- Reagir bem a cargas elevadas.

Área de Swap

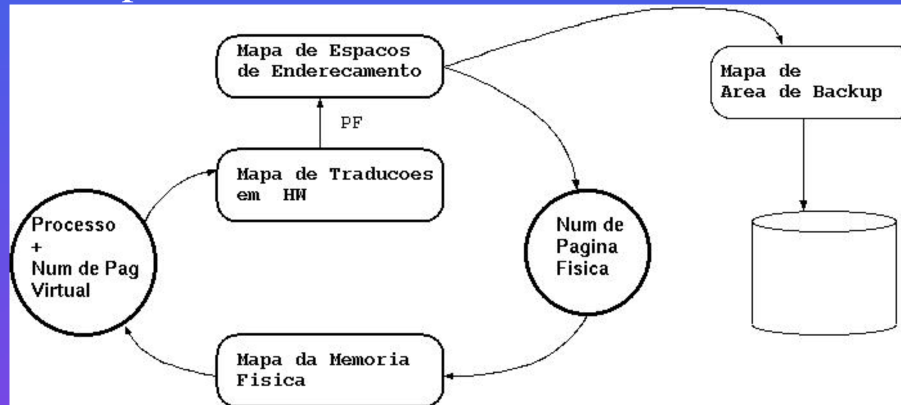


Precisamos de

- *swap-map* para descrever a posição de páginas swapped;
- Não é preciso guardar páginas de texto.

Mapas de Tradução

Precisamos de vários mapas:



- Traduções de endereços em HW: podem ser TLBs e/ou tabelas de páginas, depende do HW mas gerido por SO;
- Mapa de espaços de endereços: usado num PF para verificar se a página é válida e carregar uma tradução de HW;
- Mapa da memória física: eg, para remover páginas;
- Mapa da área de backup, que pode ser o arquivo ou swap.

Substituição de Páginas

Várias técnicas são possíveis:

- *Ideal*: página morta, que nunca mais será usada, eg, de processo terminado;
- *Local*: cada processo tem um número de páginas;
- *Global*: olhar para sistema;
- *Working Set*: páginas que vão ser precisas no futuro próximo;
- *Localidade de Referências*: conjunto de páginas mais importantes muda devagar;
- *LRU*: libertar a página acedida à mais tempo.

Suporte de HW

Unidade de Gestão de Memória (*MMU*):

- Tradução de endereços virtuais;
- Tabelas de páginas:
  - Uma para endereços de kernel;
  - Uma ou mais para cada processo;
  - Entrada (*PTE*) com 32 bits e incluem núm. de frame, info. de protecção, se *válida*, se *modificada*, se *referida*;
  - MMU usa apenas as tabelas activas;
  - Erros de tradução geram "Page Fault" (*PF*):
    - Endereço fora de limites;
    - Página inválida;
    - Erro de Protecção

Suporte de HW

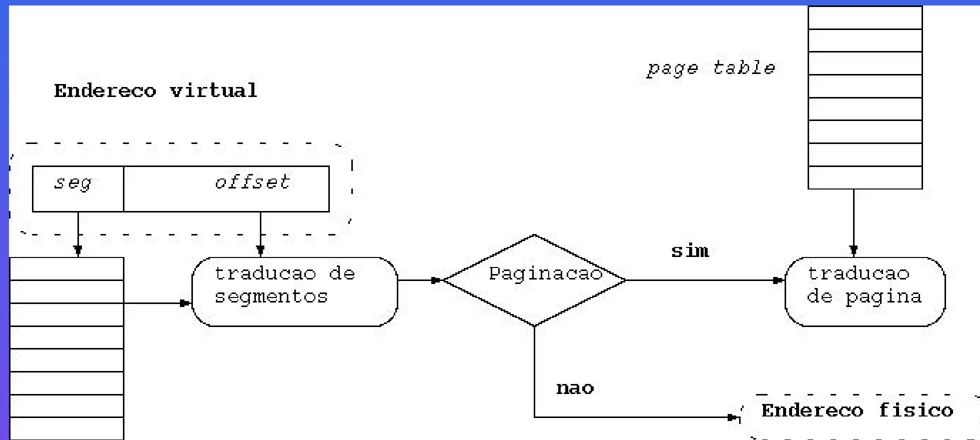
- Problema do tamanho: 512 mil entradas para 2GB e páginas de 4KB:
  - ⋆ Segmentação;
  - ⋆ Paginação
- *Context Switch*.

TLB

Cada acesso a memória obrigaria a acedar PT:

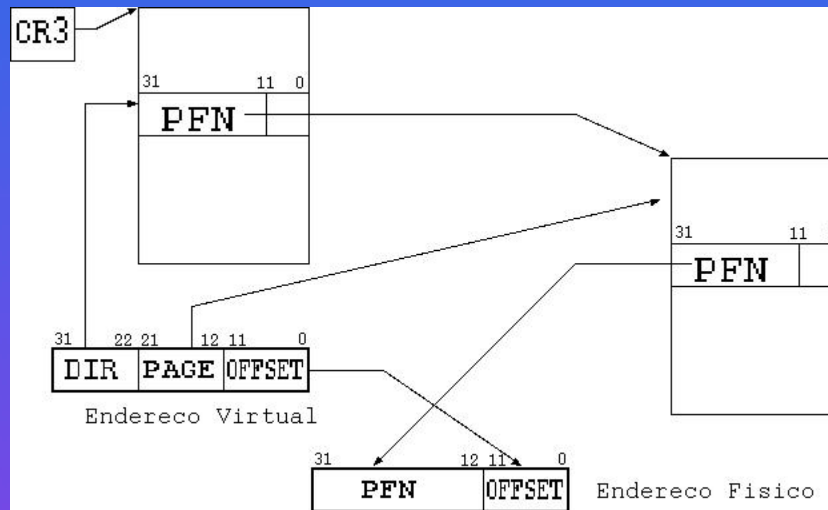
- Adicionar uma cache rápida procurada antes da PT (*L0*):
  - Endereço físico;
  - Endereço virtual (HP).
- Cache associativa com as traduções mais recentes, TLB:
  - controlada pela MMU;
  - SO tem que manter coerência se mudar PT;
  - Alterações podem ser explicitas ou side-effect de instruções;
  - Alguns sistemas só usam TLB.

Hardware *x86*



- 32 bits: 4GB de endereços;
- Paginação pode ser desabilitada usando **CR0**;
- Até 8K segmentos: *segment descriptor* descreve o segmento que é visto pela *LDT* do processo mais *GDT* global;
- Unix só usa segmentação para proteção de memória, entrada no kernel, e mudança de contexto:
  - ⋆ Todos segmentos do usuário têm base 0 e tamanho grande;
  - ⋆ seg. especiais: *call gate* para entrada no kernel e *task state segment* para contexto.

Hardware *x86: Paginação*



- Dois níveis de páginas;
- Directório de páginas: contém PTEs que mapeiam as próprias páginas (1024 de 4B);
- **CR3** ou **PDBR** armazena endereço físico de dir;
- Cada PTE contém pág física, `RW`, e se *válido*, *referido* e *modificado*;
- Escrita em **CR3** faz reset da TLB;
- 4 níveis de privilégio.

Hardware: RS/6000

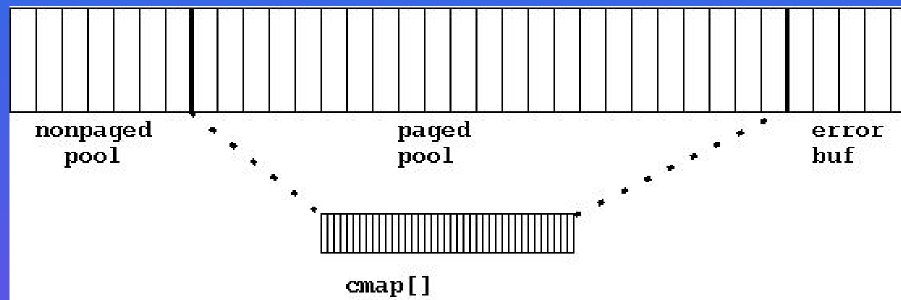
- RS/6000 usa *tabela de páginas invertida*:
  - Entrada por frame;
  - Hash é usada para encontrar página virtual;
  - Compacto, mas lento;
  - RS/6000 usa espaço de endereços de 52-bits;
  - Processo tem 16 segmentos:
    - kernel text;
    - user text;
    - dados privados;
    - dados partilhados (7);
    - dados de VM (2);
    - texto partilhado;
    - dados do kernel;
    - I/0.
  - 4 bits de cima são convertidos em 24 bits de selecção.

VM em 4.3BSD

Baseado no 3BSD:

- Originalmente para VAX-11: emulação da arquitectura do VAX em sofware;
- Estruturas:
  - *core map* descreve memória física;
  - *page tables* descrevem memória virtual;
  - *disk maps* descrevem áreas de swap;
  - `proc` e `u-area`.

Memória Física



Três áreas:

- *Nonpaged pool*: código do kernel e memória do kernel alocada estáticamente;
- *Error buffer*: mensagens de erro em crash;
- Páginas de processos e páginas de memória dinâmica do kernel (nonpageable).

Core map sobre cada frame:

1. *Nome* processo dono, tipo, e VPN:
   - dono de pág de texto é uma estrutura texto;
2. Lista de Páginas Livres
3. Cache de Páginas de Texto
4. Sincronização

Espaço de Endereçamento

Emula VAX-11:

- 4 áreas iguais,

  P0: texto e dados do processo;

  P1: pilha de kernel e usuário, `u-area`;

  S0: texto e dados do kernel;

  S1: não é usada.

- PTs com formato baseado no VAX:
  - ⋆ 1 de sistema para **S0**;
  - ⋆ cd processo tem mapa **P0** e **P1**;
  - ⋆ Contíguas em memória virtual e alocadas por mapa em *Userptmap* de **S0**;
  - ⋆ Mapa de recursos $< base, size >$ descreve área que estão usadas (first-fit);
  - ⋆ Swapper é usado para libertar entradas em *Userptmap*
  - ⋆ Partilha em **P0** é possível em regiões alinhadas com 64KB, mas complicada: partilha explicita.

Vida de Página

3 Estados:

Residente: em memória física;

Fill-On-Demand: ainda não foi referida, pode ser *fill-from-text* ou *zero-fill*;

Outswapped: recuperáveis de swap,

bits válido e fill-on-demand a 0, e PFN a 0.

Espaço de Swap

Partições cruas, sem filesystem

- Várias partições: usa *interleaving* para melhorar performance;
- Alocação de swap apenas para páginas que vão ser enviadas: reduz swap
  - ⋆ pode levar a *memory overcommit*;
  - ⋆ BSD é conservador: kernel aloca espaço necessário à partida.
- Páginas de texto não precisam de swap, dá problema:
  - ⋆ inicialmente bloco está na PTE;
  - ⋆ Reescrito com PFN quando página trazido;
  - ⋆ Páginas são guardadas em swap.
- Alocação usa estrutura `dmap`:
  - ⋆ array de tamanho fixo com chunks no swap;
  - ⋆ `u-area` guarda mapas para dados e swap;
  - ⋆ texto em text structure.

`fork()` em BSD

Gestão de Memória:

1. Alocação de *swap* para dados e pilha;
2. alocaçào de PTEs em `Userptmap`, senão swapping de outro processo;
3. Área-u é inicializada e colocada no map `Forkmap`;
4. Região de texto: filho é adicionado aos processos usando esta estrutura;
5. Dados e texto são copiados pág a pág.

Caro!!

- Copy-on-write: obriga a contadores de referência por página;
- `vfork()` apenas cria `u-area` e `proc`, muito rápido.

PF em BSD

Numa PF o sistema guarda informação de estado e chama rotina de PF:

1. Se violação de limites, `SIGSEGV` ou aumenta pilha, senão chama `pagein()`;
2. Se simulação de `referenced`, coloca `valid` a 1;
3. Se PF residente mas na free lista, `valid` a 1 e entrada `cmap` fora de free list;
4. Se pág de texto e outro processo está a ler: *locked* e *in-transit*; usa *wanted* e `sleep()`
   - segundo processo pode ter perdido a pág.
5. Procurar tabela de hash para páginas de texto que não na PTE;
6. Ler do swap se em swap;
7. Alocar e colocar a 0s para *zero-fill*;
8. Ler de executável para `fill-from-text`: primeiro na buffer-cache, se lá flush, ler de disco.

Lista de Páginas Livres

Se memória cheia, qual a melhor pág para remover?

1. Pág de processo terminados;
2. Senão usar LRU:
   - LRU completo é impossível;
   - NRU: relógio com 2 mãos, uma desactiva bit referido e outra verifica o bit.
3. kernel mantém páginas livres entre `minfree` e `maxfree` com *pagedaemon*:
   - mapeia páginas no seu espaço de endereçamento;
   - escreve directamente no swap;
   - usa write assíncronos;
   - em completion coloca na lista limpa donde são retornados para lista de memória livre.

Swapping

Sistema geralmente funciona bem, mas

- Carga pesada pode entrar em *thrashing*: não conseguimos manter todos os working sets em memória;
- Solução: desactivar processos com *swapper*:
  - ⋆ se `Userptmap` está muito fragmentado libertar um processo corrente;
  - ⋆ Se `freemem` abaixo de limites desejáveis por muito tempo;
  - ⋆ Se processo inactivo por muito tempo.
- Candidatos:
  - ⋆ processo que dorme há mais de 20 segundos;
  - ⋆ dos 4 maiores, o que está em mem. há mais tempo.

BSD: Conclusões

Boa funcionalidade e poucas exigências sobre HW, mas:

- Não há acesso a arquivos remotos;
- Não há partilha de memória;
- `vfork()` limitado;
- Cada processo com PT para texto partilhado: gasto extra e sincronização;
- Arquivos mapeados em memória e librarias partilhadas?
- Breakpoints do debugger causam confusão.
- Alocação de swap muito conservadora;
- Swap remoto;
- Influência do VAX;
- Código não é modular.

Ideal para disco grande, CPU lento, e pouca memória.

VM em SVR4/Solaris

Ideias:

- Mapeamento de Arquivos (privados e partilhados):
  - ⋆ Util para usuários;
  - ⋆ mas não substitui `read` e `write`;
  - ⋆ Fundamental no kernel.
- 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.

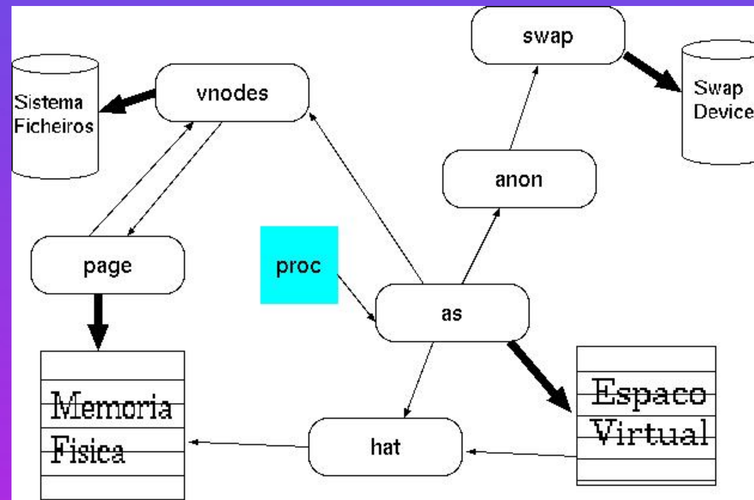
Desenho de VM em Solaris

- O espaço de endereçamento é constituido por um conjunto de *mapeamentos* para diferentes objectos.
- Cada objecto de memória é uma *sub-classe* de uma classe base.
- Cada mapeamento é uma *instância* da sua sub-classe.
- Objectos de memória podem ser associados a um nó-v (muitos para um).
- Objectos não associados a nós-v são representados por *anon*, o objecto anónimo.
- Memória é baseada em páginas.
- Páginas físicas servem como cache para objectos pageados.
- Arquitectura de VM é independente de Unix,
  componentes dependentes de HW no HAT.
- Kernel usa copy-on-write quando possível.

Abstrações Fundamentais de VM

O sistema usa

1. `page`, página;
2. `as`, espaço de endereçamento;
3. `seg`, segmento;
4. `hat`, tradução de endereços em hardware;
5. `anon`, páginas anónimas:



Página Física

Memória é dividida em regiões paginadas e não paginadas:

- Cada `page` descreve página lógica (grupo de pág físicas);
- pág tem inf sobre $< no - v, offset >$, e está na lista do nó-v: permite nome único mesmo se partilhada;
- pág em hash-chain baseada no $< no - v, offset >$;
- nó-v mantém listas de páginas;
- pág pode estar na *free list*.
- Ref. count e flags de sincronização, mais cópias dos bits *modifed* e *referenced*;
- informação do HAT usada para obter todas as traduções da página.

Espaço de Endereçamento

`proc` aponta para `as`:

- Lista de mapeamentos do processo, `seg`;
- Inclui o `hat`;
- *hint* sobre o último seg. com PF;
- Flags;
- Operações sobre `as`:
  - ⋆ `as_alloc()` dá novo `as`;
  - ⋆ `as_free()` liberta `as`;
  - ⋆ `as_dup()` duplica.
- Operações sobre grupos de páginas:
  - ⋆ `as_map()` e `as_unmap()` coloca objectos no `as`;
  - ⋆ `as_setprot()` e `as_checkprot()`;
  - ⋆ `as_fault()`, começa PF;
  - ⋆ `as_faulta()` faz *fault ahead*.

Mapeamentos

Existem vários tipos de segmentos:

- Campos públicos são base, tamanho, `as`;
- Funções virtuais em `seg_ops`:
  - ⋆ `dup()` duplica mapeamento;
  - ⋆ `fault()` e `faulta()`;
  - ⋆ `setprot()` e `checkprot()`;
  - ⋆ `unmap()` termina;
  - ⋆ `swapout()` do swapper;
  - ⋆ `sync()` sincroniza.
- Existe ainda `create()` que é chamado da função genérica para alocar o segmento.
- Outras funções genéricas: libertar, *attach* e *unmap*.

Páginas Anónimas

Criada quando processo modifica página `MAP_PRIVATE`:

- Modificações não afectam o objecto original;
- Páginas inicializadas tornam-se anónimas quando são modificadas;
- Armazenadas em swap;
- Objecto anónimo:
  - único no sistema;
  - representação: v-nó `NULL`;
  - usa swap para backup storage;
- refs a páginas são contadas;
- `anon` exporta `dup`, `free`, `private` (cópia privada), `zero` e `getpage`.

O HAT

Operações dependentes do HW são isoladas no HAT. É acedido por uma interface:

- HAT: alocação, dealocação, dup, swapin e swapout.
- conjunto de páginas: `hat_chgprt()`, `hat_unload()`, `hat_memload()` e `hat_devload()` (últimos para uma única página).
- todas as traduções de uma página: `hat_pageunload()`, `hat_pagesync()`.

Informação no HAT é redundante e altamente dependente do HW:

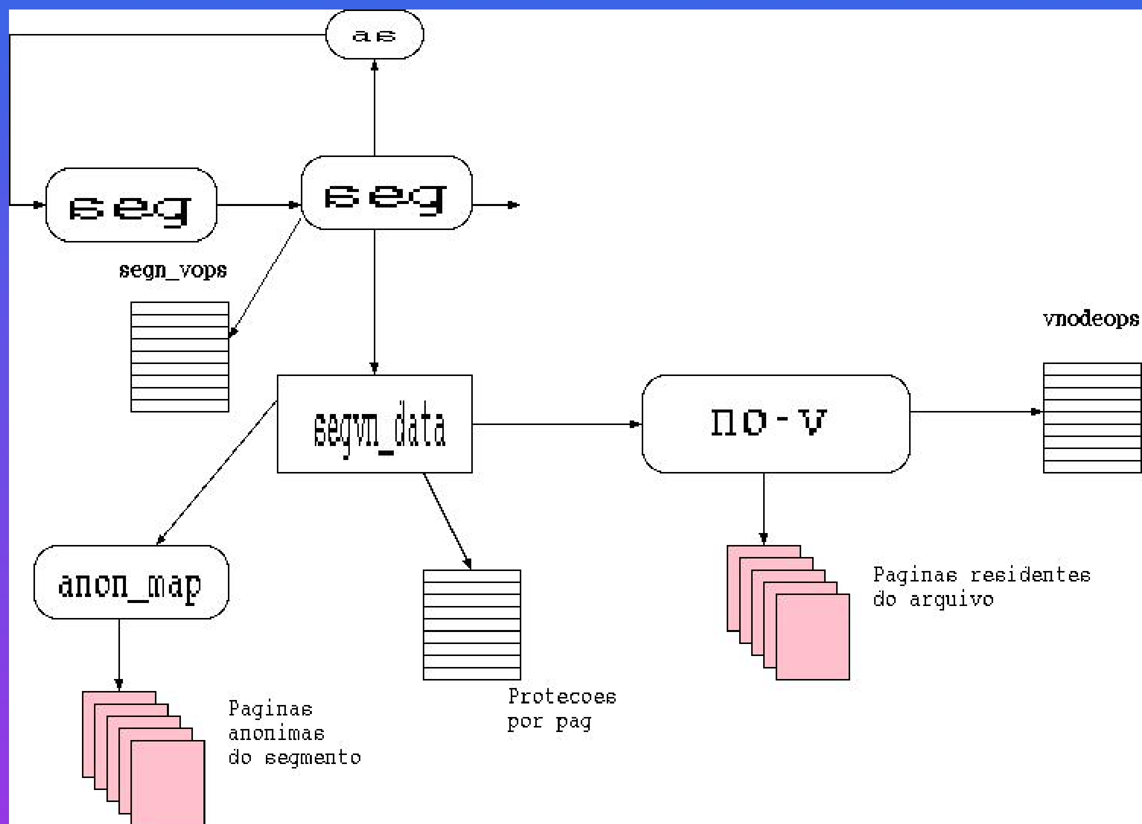
- Traduções para página mantidas em lista (`hat_unload()`);
- Ref. x86;
- Tradução é chamada de *chunk*.

Drivers de Segmentos

Existem diferentes tipos de segmentos:

- `seg_vn`: endereços para ficheiros regulares ou anónimo;
- `seg_map`: permite fazer acessos do tipo `read` e `write`;
- `seg_dev` mapeia devices como frame buffers, memória.
- `seg_kmem` mapeia regiões do kernel em memória alocada não dinâmicamente.

seg_vn



Mapeia:

- nó-v, ou,
- anónimos: NULL, `/dev/zero`

seg_vn em detalhe

Dados incluem:

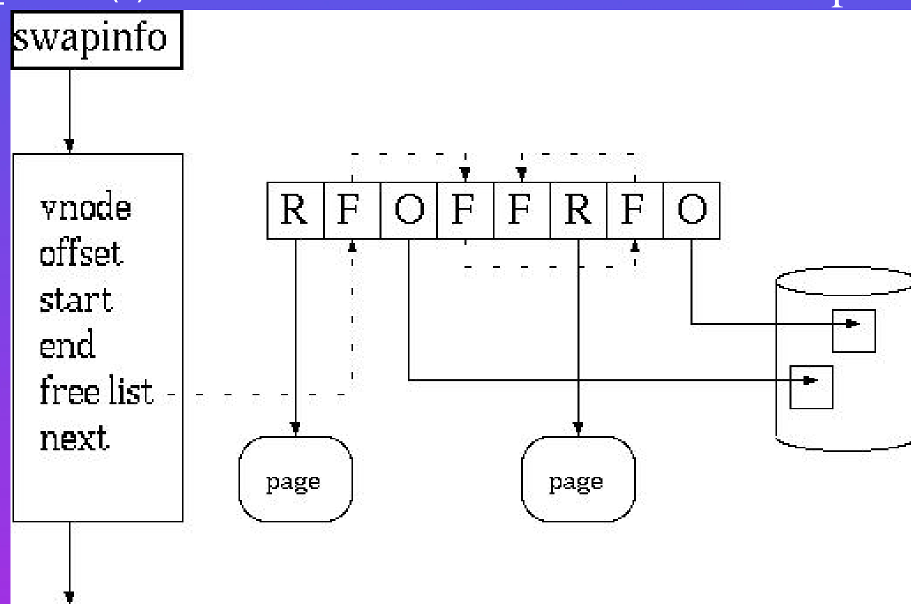
- proteções correntes e máximas;
- tipo de mapeamento (partilhado ou privado);
- ptr para nó-v;
- offset do segmento no arquivo;
- ptr para mapa anónimo;
- ptr para um array de protecções por página, se pags com protecções diferentes.

Protecão máxima é a inicial.

Swapping

Responsabilidade do *anon layer*:

- Rotina `swap_xlate` mapeia estruturas tipo anon e páginas outswapped.
- Dispositivos de swapping podem ser colocados e removidos dinâmicamente do sistema por `swapctl()`. Cada um deles tem uma estrutura tipo `swapinfo`.



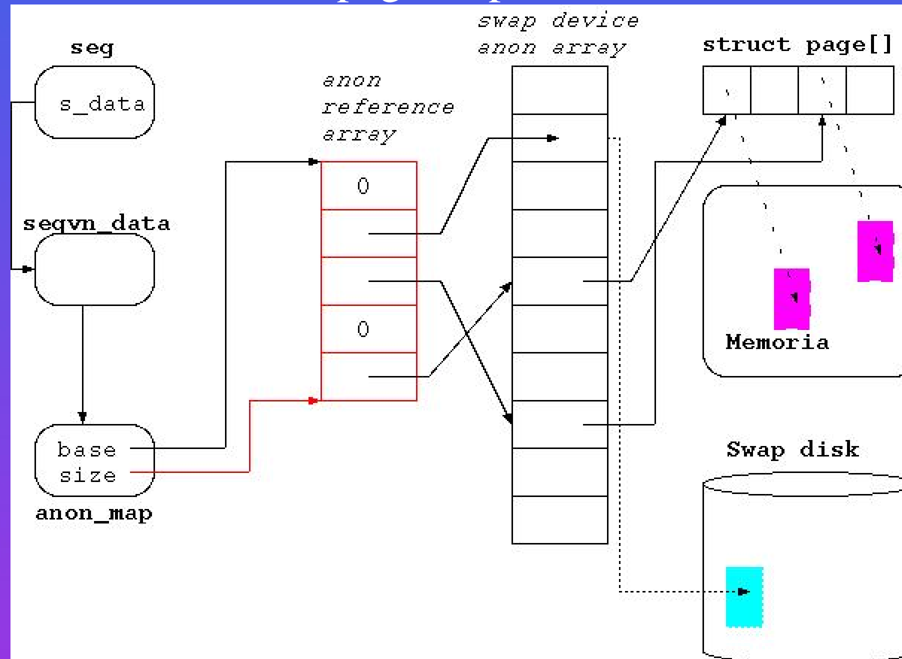
- Segmentos alocam todas as páginas de swap que vão precisar antes de começar.

Mapeamento Novo

- 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 mapas 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.



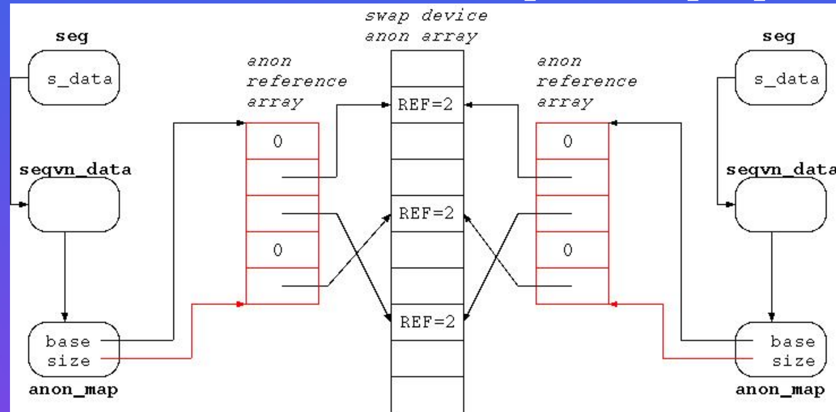
- anon array inclui núm de referências. Pág. são removidas quando núm. de ref. desce a zero.
- As estruturas de dados são criadas na primeira escrita a pág. privada.

Novo Processo

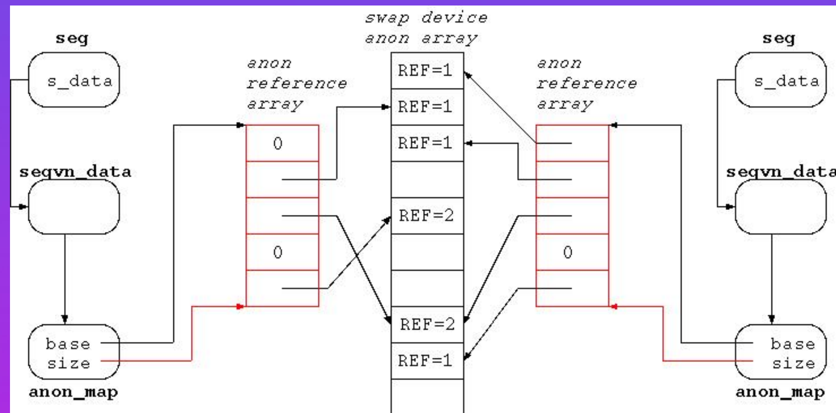
- `fork()` chama `as_dup()` para duplicar o espaço do pai;
- `as_dup()` primeiro chama `as_alloc()` e chama o `dup()` de cada segmento;
- Duplicação aloca um novo `struct seg`, campos são copiados do pai:
  - ⋆ mapeamntos para texto, pilha e dados são colocados a `MAP_PRIVATE` no pai e filho. `MAP_SHARED` continuam.
- `hat_chgprot()` protege contra escrita todas as págs anónimas;
- `anon_dup()` duplica `anon_map`: copia todos os ptrs no array e incrementa ref counts;
  - ⋆ `hat_dup()` duplica `hat` e informação de tradução.
  - ⋆ `hat_dup()` pode alocar novas PTs.

Copy On Write

Uso de `swap device reference array` permite que partilha seja por página:



Depois de PF:



Gestão de Páginas Livres

pagedaemon implementa relógio:

- mão da frente usa `hat_pagesync()` para remover referência e alterar `hat`;
- páginas sujas são enviadas para disco por `VOP_PUTPAGE()`, que implementa *clustering*;
- `hat_pageunload()` invalida páginas que saem para lista livre
- Bit de referência pode ter que ser simulado.

Swapping

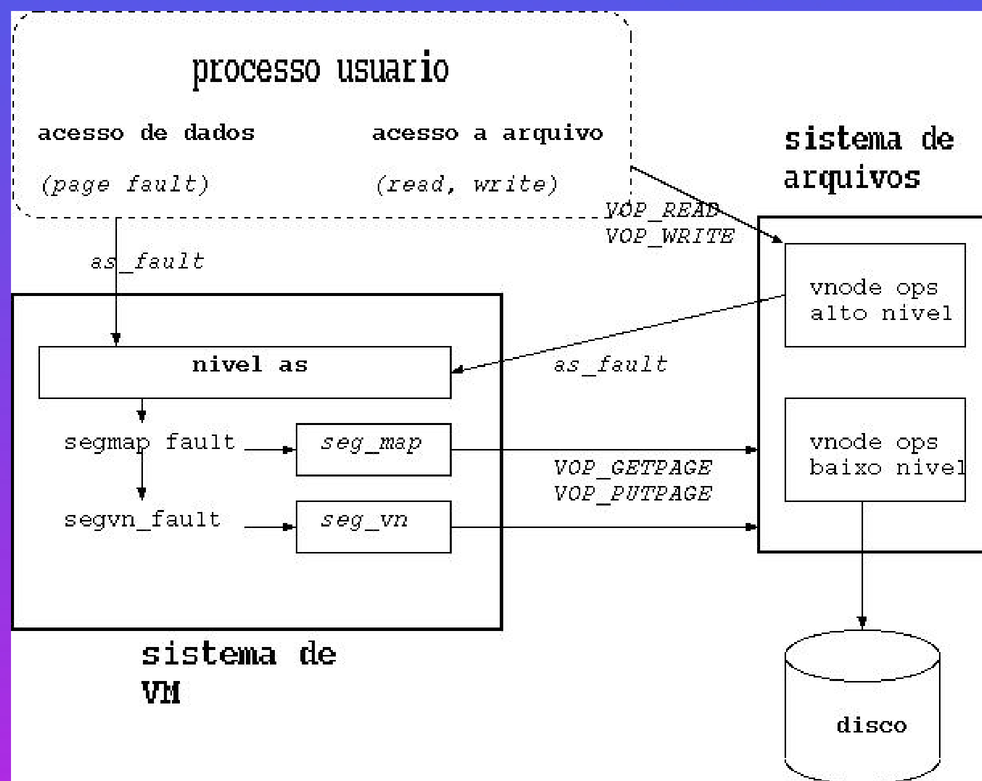
Processo `swapper` tem PID 0 e é chamado cada segundo:

- se memória livre < `t_gpgslo`, envia processo para fora usando `CL_SWAPOUT();`
- chama `as_swapout()` para enviar um processo:
  - chama `swapout()` por segmento;
  - maioria dos segmentos usa `segvn_swapout()`.
- No fim faz swap de `u_area`;
- Swapin: basta trazer `u_area`.

VM e Sistema de Arquivos

Relação simbiótica:

- VOP_GETPAGE( ) é chamada para obter páginas;
- VOP_PUTPAGE( ) é chamada para enviar páginas sujas;



Vantagens

- Desenho muito modular, OO;
- Isolar HW no `hat`;
- suporta COW, `MAP_SHARED`, arquivos.
- suporta `mmap()`;
- suporta bibliotecas partilhadas;
- beneficia do nó-v;
- Unifica buffer-cache com VM cache;
- breakpoints funcionam pq texto é `MAP_PRIVATE`.

Desvantagens

- Mais informação (40B por pág vs. 16B);
- Páginas de arquivos não são guardadas em swap: procura mais lenta;
- alg. mais complexos e mais lentos;
- não computa endereço no disco durante `exec()`: procura mais lenta;
- Tuning é mais dificil por ser OO;
- COW pode ser mais lento que cópia puura;
- Swap é alocado por página, perde clustering.

Melhoramentos

Tipos de modificações:

- Válido para inválido: propagação imediata;
- Inválida para válida: 2 processadores partilham a pág,
  - ⋆ preguiçoso: segundo processo não vê transição;
  - ⋆ vs. propagação imediata.
- Resulta em muitos PFs, e mais lento com OO;
- COW pode ser caro: 1 em 4 págs têm que ser copiadas;

Número de PF grande na versão inicial.

Optimizações

Alguns melhoramentos:

- Inicializar mapas de tradução em `fork()`:
  - filho executa sempre algum código
- Inicializar parte dos mapas de tradução depois de `exec()`:
  - para cada nó-v, inicializar as entradas para páginas do nó-v em memória;
  - é estimar o conjunto de trabalho como o conjunto de pág em memória
- Estudar o comportamento de `sh` e amigos para copiar algumas páginas imediatamente.

VM no Linux

- Páginas são inicialmente COW ou zero-fill;
- `vm_area` descreve um segmento;
- Linux assume tabela de três níveis:
  - ⋆ Tabela de pág descreve `vm_obj`;
  - ⋆ `fork()` copia PTEs,
  - ⋆ páginas swapped out são representadas na PTE,
  - ⋆ Não pode fazer swap out de PTs,
  - ⋆ Toda a memória física é colocada em KVM:
    1. evita indirecções,
    2. problemas com configurações grandes;
- em x86 tabela do meio tem 1 entrada!

PFNs no Linux

- `mem_map_t` descreve página física:
  - `count` é o número de usuários;
  - `age` é a idade;
  - `map_nr` é o PFN.
- Alocação a partir de `free_area`

LRU no Linux

- `lru_page` aponta para a cabeça da fila;
- quando pág é libertada é colocada no fim da fila;
- `shrink_mmap()` tenta libertar páginas;
- Usa um bit de referência para manter idade:
  1. Idade inicial é 3;
  2. Quanto pág é tocada pelo MM idade aumenta de 3;
  3. Envelhecimento por `swap` decrementa de 1

Swapping em Linux

`kswapd` mantém memória livre:

- Geralmente na fila `kswapd_wait`;
- Acordado pelo alocador quando memória livre desce abaixo de certo valor:
  1. Tenta libertar páginas livres;
  2. Tenta diminuir `dentry`;
  3. Tenta diminuir `inode`;
  4. Tenta colocar memória partilhada fora;
  5. Tenta libertar páginas sujas.
- Gestor de VM tenta não escrever muitas pág no swap ao mesmo tempo;
- Alocação de swap é linear.