Resumos SO
1º Teste

by Fox

____Processos

O **Modo Utilizador** separa-se do **Modo Kernel** por meio de uma barreira de protecção que só é quebrada aquando invocada uma interrupção de software (System Call). Uma aplicação em modo de utilizador não pode suprimir/alterar registos, situação crítica a nível de segurança

Pseudoparalelismo ou pseudoconcorrência – implementação de sistemas multiprogramados sobre um computador com um único processador.

Programa - Ficheiro executável sem actividade
Processo - Objecto do sistema operativo que suporta a execução dos programas.

Um **processador / processo** tem:
– Espaço de endereçamento (virtual):
• Conjunto de posições de memória acessíveis
• Código, dados, e pilha
• Dimensão variável
– Reportório de instruções:
• As instruções do processador executáveis em modo utilizador
• As funções do sistema operativo
– Contexto de execução (estado interno):
• Valor dos registos do processador
• Toda a informação necessária para retomar a execução do processo
• Memorizado quando o processo é retirado de execução

Criar processo: IdProcesso = CriaProc (Código, Prioridade,...)
Eliminar processo: EliminarProc (IdProcesso)
Bloquear processo: Estado = EsperarTerminacao (Idprocesso)

Modelo de segurança – Consiste em associar um processo a um UID(responsável pelas acções) e a um GID (para facilitar partilha).

PID – Process ID
PPID – Parent Process ID

id = fork()
Cria um novo processo cópia do processo Pai, mas com PID diferente.

void exit (int status)
Termina o processo devolvendo o estado em que o processo terminou para o processo pai.

int wait (int *status)
Bloqueia o processo pai até que **um** dos filhos termine.

int execv(char* ficheiro, *argv [])
permite substituir a imagem do processo
onde é invocada pela contida num ficheiro executável. Mantem o PID.

____Tarefas

Tarefas - Mecanismo simples para criar fluxos de execução independentes, partilhando um contexto comum.

thread_create - Cria nova thread
thread_yield – Faz com que a thread deixe o CPU para outra thread poder correr.
thread_wait – Permite esperar que uma thread específica termine
thread_exit –Destroi uma thread

Porque usar tarefas em vez de processos?
-As tarefas são entidades paralelas podem partilhar o mesmo espaço de endereçamento e todos os dados que ele contem.
-Visto não terem recursos individuais é muito mais facil criar e destruir threads do que processos.
-Embora não se tenha grandes vantagens se todas as threads usarem o CPU, se for preciso lidar também com I/O as tarefas podem-se sobrepor, resultando em muito mais desempenho.

Visto que as tarefas partilham o mesmo espaço de endereçamento têm acesso às mesmas variáveis globais. A modificação e teste das variáveis globais tem de ser efectuada com precauções especiais para evitar erros de sincronização.

Pseudotarefas (Tarefas-Utilizador) -Tarefas implementadas numa biblioteca de funções no espaço de endereçamento do utilizador. A lista de tarefas e o contexto são mantidas no processo que lançou as threads.

Tarefas-Núcleo (ou Tarefas Reais) - Implementadas no núcleo do SO. A lista de tarefas e respectivo contexto são mantidos pelo núcleo.

Comparação Tarefas Utilizador e Núcleo

- **Capacidade de utilização em diferentes SOs?**

- As threads-nucleo não podem ser usadas se o SO não suportar. Com as threads-utilizador, podem-se implementar threads num SO que não as suporte.

- **Velocidade de criação e comutação? (vs.processos?)**

- As threads-utilizador são mais rápidas do que as threads-nucleo, sendo que os processos são os mais lentos que ambos

- **Tirar partido de execução paralela em multiprocessadores?**

- em multiprogramação, lança uma thread-kernel por processador, bem como nos processos

- **Aproveitamento do CPU quando uma tarefa bloqueia (ex: ler do disco)?**

- Nas threads-utilizador quando alguma bloqueia (fazendo um system-call) todas as outras threads ficam bloqueadas até a situação ser resolvida. As thread-kernel tem maior aproveitamento, porque quando uma bloqueia o SO pode correr outra thread do mesmo processo ou até de um processo diferente, se tal for necessário.

____Eventos (Signals)

Rotinas assincronas - Certos acontecimentos que devem ser tratados pelas aplicações,embora não seja possível prever a sua ocorrência.

Poder-se-ia dedicar-se uma tarefa à espera de uma acontecimento, mas isso teria a desvantagem de implicar a existência de testes sistemáticos, que ocupavam ciclos de CPU.
Em alternativa têm-se as **rotinas assíncronas** associadas aos acontecimentos (eventos ou signals)

Tratamento dos Signals
• **Por omissão** – termina o processo.
• **Ignorado** – Alguns signals como o SIGKILL não podem ser ignorados.
• Associado a uma rotina de tratamento (**handler**) através da chamada à função sistema signal.

signal (Sinal a captar,Procedimento Handler para Sinal)

A função **kill** envia um sinal ao processo. Apesar do nome, pode não matar o processo desde que o signal tenha um tratamento associado ou esteja ignorado.

____Gestão de Processos

Gestor de processos – A entidade do núcleo responsável por suportar a execução dos processos. Faz a multiplexagem do processador(**despacho** e **escalonamento**), faz a **gestão das interrupções** e encarrega-se das **funções de sincronização**.

As **System Calls** podem ser provocadas por : **Hardware** (relógio ou periféricos), **Software** (traps, software interrupts) ou **Excepções** (provocadas pelo programa em execução como divisão por zero ou acesso a memória indevido).

As **System Calls** funcionam do seguinte modo:

1- **Interrupção** (salvaguarda contexto na pilha actual)
2- **Gestor das Interrupções** (identificação da interrupção -- vector de int.)
3- **Rotina de Serviço de Interrupção** (corre o código especifico à interrupção, possivelmente alterando o estado dos processos)
4- **Despacho** (eventualmente, para escolher outro processo).
5- **Retorno da Interrupção**

Despacho - Tem como função comutar o processador sempre que lhe seja indicado para o fazer.

Funciona da seguinte maneira:

1 - Copia o contexto hardware do processo em execução da pilha actual para o respectivo descritor (entrada na tabela de processos)
2 - Escolhe o **processo mais prioritário** entre os executáveis
3 -Carrega o contexto hardware no processador
4 - Transfere o controlo para o novo processo

Função de Escalonamento - Define quem deve ser o **próximo processo** a executar-se, de acordo com a política de escalonamento, com vista a **optimizar** a utilização do processador (e dos restantes componentes dosistema). Deve ser invocada, **em teoria**, sempre que um recurso do sistema é atribuído ou libertado. O problema disso é que demora tempo.

Políticas de Escalonamento:

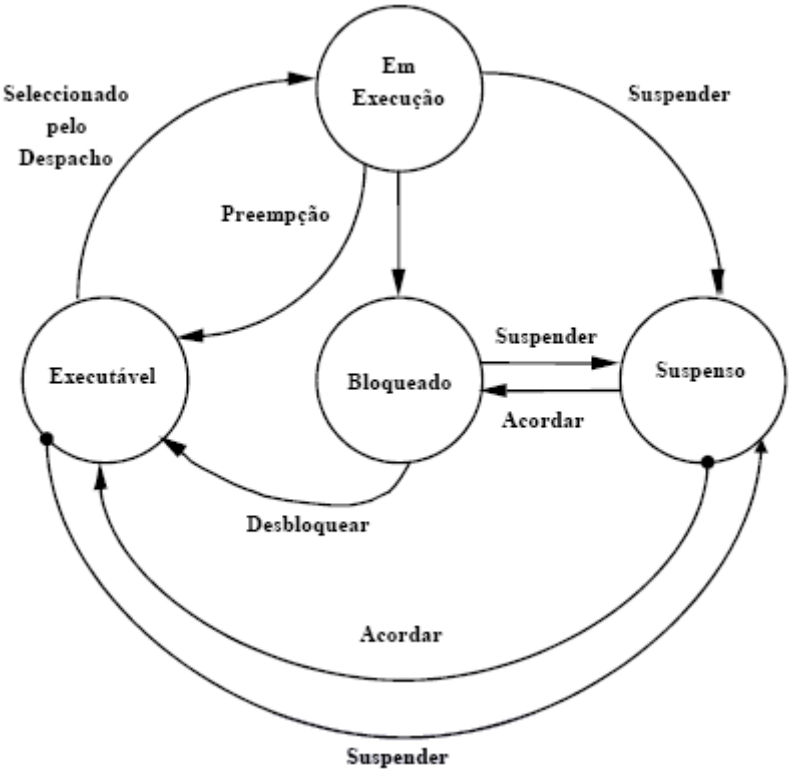
→**Time-Slices (ou Round-Robin)**
Tempo de execução de um processo é limitado a um **quantum de tempo** (**time-slice**), de modo a permitir que todos os processos executáveis tenham oportunidade de dispor do processador ciclicamente.
A **lista de processos** executáveis é gerida em **round-robin**, mas pode conduzir a tempos de resposta elevados em situações de muita carga.

→O valor do **quantum** deve ser **aumentado** quando o sistema está **muito carregado**, com vista a limitar o custo dos da comutação de contextos e aumentar a probabilidade do processo terminar.

→**Prioridades** - Permitem definir a importância de um processo no processo de escalonamento. Podem ser **Fixas** (processos de tempo real) ou **Dinâmicas** (nos sistemas de tempo virtual, priveligiando I/O), mudando consoante o comportamento do processo.

Um processo deve ser **promovido** quando é **bloqueado** e **relegado** quando **termina o seu time-slice**.

→**Preempção** – Acção de retirar o processador a um processo em execução devido à existência de outro mais prioritário. Permite que os processos mais prioritários reajam rapidamente a um dado acontecimento (reactividade aos acontecimentos externos).



Um processo fica **Bloqueado** quando é efectuado, por exemplo, um System Call.
Um processo fica **Supenso** por acção de sincronização (p.ex: Semáforos).

____Gestão de Processos em UNIX

Um processo em **Modo Utilizador** executa o programa que está no seu segmento de código e muda para **Modo Sistema** sempre que uma **excepção ou interrupção** é desencadeada, podendo essas ser provocada pelo utilizador ou pelo hardware (como já foi visto antes).

A mudar para **Modo Sistema** corresponde a:
– mudança para o **modo de protecção** mais privilegiado do processador;
– mudança para o **espaço de endereçamento** do núcleo;
– mudança para a **pilha núcleo** do processo.

A **Pilha Núcleo**:
É usada a partir do instante em que o processo muda de modo utilizador para modo núcleo, caso contrário está vazia.
É necessária para garantir a estanquicidade de informação entre a actividade das funções do núcleo.

Escalonamento

→Em **Modo Utilizador**

O escalonamento é **preemptivo**, efectuando trocas entre processos, sendo que as prioridades são calculadas **dinamicamente** (e periodicamente recalculadas) em função do tempo de processador utilizador.

Quando o processo passa a **Modo Nucleo** (quando **bloqueia** e é feita um **System Call**) é lhe atribuída uma prioridade em **Modo Núcleo**, de acordo com o recurso que detém aquando do bloqueio.

As prioridades são calculadas utilizando a seguinte formula :

Prioridade = TempoProcessador/2 + PrioridadeBase

→Em **Modo Núcleo**

Os processos em modo núcleo não é comutados e as prioridades são **fixas** e definidas com base no acontecimento que o processo está a tratar.Sendo sempre superiores às prioridades em Modo Utilizador.

Alguns processos (os mais criticos) do Modo Núcleo nunca podem receber Sinais.

Quando passa para **Modo Utilizador** o sistema **recalcula** a prioridade do processo e atribui-lhe uma prioridade em **Modo Utilizador**.

Existe, no entanto, um problema com algoritmo de escalonamento do Unix que diz respeito à **Escalabilidade** (capacidade de suportar uma **quantidade crescente** de trabalho de forma uniforme) porque o tempo em UNIX está dividido em épocas e cada época termina quando todos os processos usaram o seu quantum.

Em **Unix** os processos mais prioritários são escolhido em primeiro lugar, o que melhora a escalabilidade visto que são mais essenciais de se executarem.

Escalonamento em "real-time" – Consiste em definir prioridades estáticas superiores às dinâmicas (em Modo Utilizador, obviamente), criando uma classe **"Real Time"** (que não corresponde a um sistema de Tempo Real). Para tal são necessárias premissões.

Despacho - É invocado quando o processo em execução não pode continuar (**bloqueou** ou **terminou**) ou quando o processo retorna ao modo utilizador.

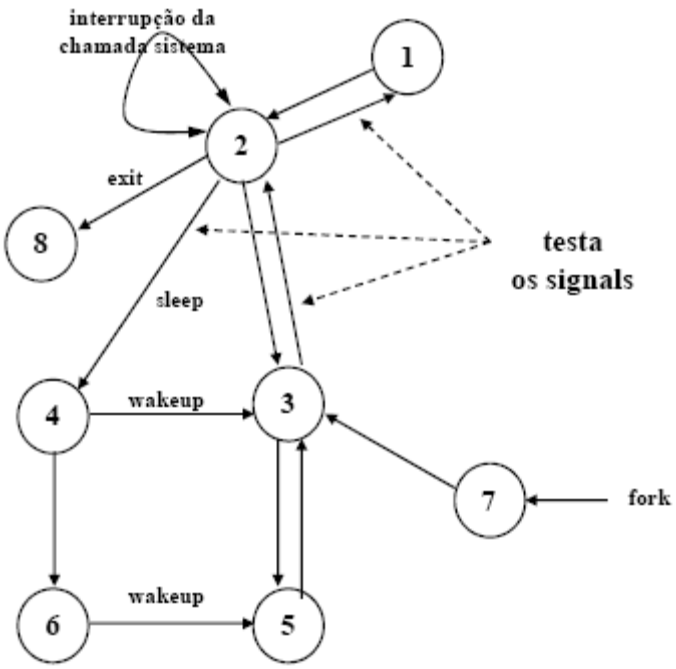
Escalonamento: Chamadas Sistema

nice (int val)

Decrementa a prioridade "val" unidades. Apenas superutilizador pode invocar com "val" negativo

int getpriority (int which, int id)

setpriority (int which, int id, int prio)



1 execução em modo utilizador
2. execução em modo núcleo
3. executável
4. bloqueado
5. executável memória secundária
6. bloqueado memória secundária
7. criação
8. zombie

Criação de um Processo

1 - Reservar uma entrada na tabela **proc** e verificar se o utilizador não excedeu o número máximo de subprocessos;
2 - Atribuir um valor ao pid (incrementando-o);
3 - Copiar a imagem do processo pai:
- Dado que a região de texto é partilhada, basta incrementar o número de utilizadores que acedem a essa região;
- As restantes regiões são copiadas;

4 - Retornar o valor do pid para o processo pai, zero para o processo filho, colocando os valores apropriados nas respectivas pilhas).

Função exit (**não** elimina o Processo)

1 – Fechar todos os ficheiros;
2 – Libertar directório corrente e regiões de memória;
4 – Actualizar ficheiro com registo da utilização do processador, memória e I/O;
5 – Enviar signal death of child ao processo pai (ignorado normalmente,);
6 – Registo proc / task mantém-se no estado zombie (permitindo ao processo pai encontrar informação sobre o filho quando executa wait).

Função wait

1 – Procura filho zombie;
2 – Pid do filho e estado do exit são retornados através do wait;
3 – Liberta a estrutura proc do filho;
4 – Se não há filho zombie, pai fica bloqueado.

Execução de um Programa
1 – Verifica se o ficheiro existe e é executável;
2 – Copia argumentos da chamada a exec da pilha do utilizadorpara o núcleo (pois o contexto utilizador irá ser destruído);
3 – Liberta as regiões de dados e pilha ocupadas pelo processo e eventualmente a região de texto (se mais nenhum processo a estiver a usar);
4 – Reserva novas regiões de memória;
5 – Carrega o ficheiro de código executável;
6 – Copia os argumentos da pilha núcleo para a pilha utilizador.

O processo fica no estado executável e o contexto núcleo mantém-se inalterado: identificação e ficheiros abertos.

Sincronização Interna
Objectos de sincronização do núcleo (p.ex: para bloquear processo num semáforo ou à espera do disco)
sleep_on - bloqueiasempre o processo
wake_up - desbloqueia todos os processos

→Signals

Envio de um signal:
O sistema operativo coloca a 1 o bit correspondente ao signal, este bit encontra-se no contexto do processo a quem o signal se destina.O número de vezes que um signal é enviado não é guardado.

Encontrar signals
– No Unix verifica-se se há **signals**, quando o processo passa de modo núcleo para modo utilizador ou quando entra ou sai do estado bloqueado;
– Em Linux é verificado quando processo comuta para estado **EmExecução**;

Tratamento do signal
1 - No descritor do processo encontra-se o endereço da rotina de tratamento de cada signal.
2 - A pilha de modo utilizador é alterada para executar a função de tratamento do signal.
3 - A função de tratamento executa-se no contexto do processo que recebe o signal como se fosse uma rotina normal

____Gestão de Memoria

Segmentação
Vantagens:

- Adapta-se à estructura lógica do programa
- Permite realização de sistemas simples sobre hardware simples
- Eficiencia nas operações que agem sobre segmento inteiro

Desvantagnes:

- Programador tem que ter algum conhecimento
- Algoritmos complicados em sistemas sofisticados
- Tempo de transferência dos Segmentos muito grandes de e para o disco é incomportavel
- Dimensão máxima limitada

➔ Ideal para Hardware simples
➔ Fragmentação Externa, visto que as partições têm tamanho variável
➔ A partilha de memória é mais facil, basta colocar nas tabelas de Segmentação dos processos o endereço real do segmento a partilhar

Paginação
Vantagens:

- Programador não tem que se preocupar
- Algoritmos simples e eficientes
- Tempo leitura disco reduzido
- Dimensão de memória virtual ilimitada

Desvantagens:

- Necessitava de Hardware mais complexo (antigamente)
- Operações mais complexas e menos elengantes
- Faltas de páginas geram sobrecarga de processamento

➔ Fragmentação Interna, visto que as partições têm tamanho fixo

Algoritmos de Gestão de Memória
Reserva (Onde colocar a informação ?)

→Paginação
Qualquer página livre
→Segmentação
Qualquer que seja o algoritmo a lista é percorrida sempre pelo menos 1 vez

Best-Fit (Menor Possível) – Gera muitos fragmentos pequenos – A procura percorre normalmente pelo menos metade da lista	**First-Fit (Primeiro Possível)** – Perde-se menos tempo com a procura, mas é gerada muito mais fragmentação – Ficam muitos blocos pequenos no inicio
Worst-Fit (Maior Possível) – "Destroi" blocos de grandes dimensões, que podem vir a ser necessários	**Next-Fit (Segundo Possível)** – Espalha blocos pequenos por toda a memória

Buddy
– Vai pertindo a memória em blocos de 2^k bytes até partir um bloco que é demasiado pequeno. Usa então o bloco imediatamente anterior. – Obtem-se um bom equilibrio entre tempo de procura e fragmentação interna e externa.

Transferência (Transferir bloco de memória secundária para memória primaria ou vice versa)

Pode ser feita:

- **A pedido** – Determinado pelo SO
  - Usada em Memória Segmentada
- **Por necessidade** – Quando se gera uma falta (de Segmento ou Página)
  - Usado em Memória Paginada
- **Por antecipação** – Preve-se que o bloco vai ser necessário

Critérios de Transferência (usados também na Substituição de Segmentos)

- **Estado e prioridade do processo** – Os processos bloqueados e com pouca prioridade são os alvos principais;
- **Tempo de permanência na memória principal** – Têm que estar sempre algum tempo em memória
- **Dimensão**

Substituição (Quando não existe memória livre qual é o bloco a retirar da memória principal ?)

Páginas

Óptimo – Retira página cujo próximo pedido seja mais distante no tempo – Usado para benchmark	**LRU (Least Recently Used)** – Menos usada recentemente – Através de contador na página que vai sendo incrementado
NRU (Not Recently Used) – Não usada recentemente – Agrupamento das páginas em 4 grupos e as menos prioritárias são removidas primeiro Grupo 0: (R=0, M=0) Grupo 1: (R=0, M=1) Grupo 2: (R=1, M=0) Grupo 3: (R=1, M=1) R -> Referênciado M-> Modificado	**FIFO (First In First Out)** – Primeiro a entrar primeiro a sair – Não tem em conta a utilização das páginas

Working Set - Conjunto de páginas acedidas pelo processo no intervalo de tempo de um processo.

Para se conseguir melhor resultado deve-se equilibrar entre a quantidade de memória que está carregada e as faltas. Demasiada memória carregada não oferece vantagem para o espaço ocupado e pouca gera muitas faltas.
Quando se muda de rotina o Working Set tem um pico e depois estabiliza.

____Sincronização

Propriedades da Secção Crítica:

- **Exclusão mútua e progresso**(**liveness**) – Protecção da mudança de variáveis numa região critica
- **Ausência de interblocagem** (**deadlock**) – Quando dois processos ficam á Espera um do outro, ficando bloqueados.
- **Ausência de míngua** (**starvation**) – Quando um processo nunca consegue aceder ao CPU (ex: Readers/Writers)

Os Algoritmos de **Peterson e Barkery** são soluções algoritmicas para os problemas de sincronização. Garantem exclusão mútua mas baseiam-se em Espera Activa.

Trincos

Fechar(Mutex);	**Bloqueia** o acesso à secção crítica.
Abrir(Mutex);	**Liberta** a secção crítica.

Semáforos

Esperar(s) Down(s)	**Bloqueia** se o valor do semaforo for menor ou igual a 0. Se **NÃO** for decrementa o contador e continua.
Assinalar(s) Up(s)	Se houver processos bloqueados **liberta** 1 deles. Se **NÃO** incrementa o contador.