

Sistemas Operativos

LEI

2023/2024

fsm@di.uminho.pt

Objectivo

- **Ajudar a perceber como funcionam os computadores**
 - Em termos físicos, o que é uma aplicação informática?
 - Que *recursos* necessita?
 - Como devem ser geridos esses recursos?
 - Como interage esta aplicação com outras?
 - Isto está lento. Que devo fazer?
 - O sistema bloqueou! Perdi tudo?
 - E muito mais...

Numa palavra...

Se fosse necessário dizer de que trata esta UC

- Numa só palavra: **CONCORRÊNCIA**
- Em mais do que uma:
 - Concorrência, concorrência...
 - Eficiência, rapidez, segurança, etc.
 - Boa gestão de recursos perante determinada carga
 - Conhecimento do funcionamento de sistemas (Apps, SO, HW)
 - Aplicações concorrentes, naturalmente!

Programa

- Recapitulação de conceitos de **programação de sistemas**
- Gestão de processos, memória, ficheiros, periféricos
- Alguma **programação concorrente** (de baixo nível)
- E mãos na massa:
 - Aulas práticas em ambiente Linux
 - (quase) nada de janelas, nem ratos...
 - terminal com *Bash*, comandos, pipelines...
 - sem internet: basta consultar o manual
 - Programação de “baixo nível”: C, syscalls, libs ...

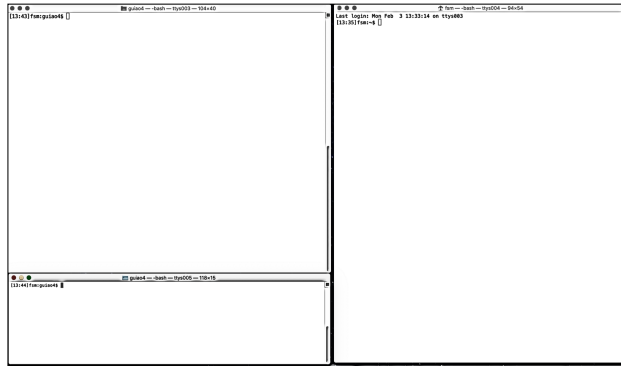




GSD

Grupo de Sistemas Distribuídos

Ambiente de trabalho: **bash + ...**



GSD

Grupo de Sistemas Distribuídos

Bibliografia recomendada

- Silberschatz, Galvin and Gagne, *Operating System Concepts*, 10th Ed, John Wiley & Sons, 2021.
- [Operating Systems: Three Easy Pieces](#), Remzi & Andrea Arpaci-Dusseau, 2018.
- **FSM 2004, Vou Fazer Exame de Sistemas Operativos!**



GSD

Grupo de Sistemas Distribuídos

Bibliografia Adicional

- Man !!!!!!!!
- R. Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, 1990.
- Beej Guides
- Google, Youtube, slashdot...



GSD

Grupo de Sistemas Distribuídos

Slides

- Baseados nos slides originais dos livros recomendados (em especial Silberschatz), caso seja necessário identificar o respectivo capítulo
- Disponíveis no Blackboard, mas em permanente revisão
- Servem apenas de “âncora” ao estudo
Não chegam para responder aos testes!
Faltam os porquês, e.g. porquê usar “isto” com esta “carga”?



Aulas práticas

- Cada aula prática tem um “guião” muito detalhado.
 - Normalmente só duram uma semana
 - Fazem sempre falta para a aula seguinte (+ trabalho prático) →
 - Sempre que resolver uma alínea, deve parar e perguntar:
O que é que EU aprendi com este exercício?
- Recomenda-se:
 - estudar o guião antes da respectiva aula
 - usar a aula para tirar dúvidas e não para colecionar resoluções; tente entender o raciocínio para lá chegar.
 - terminar em casa todas as alíneas não resolvidas durante a aula; se necessário, peça ajuda por mail



Avaliação

- **Trabalho prático** em grupo de 3, + um **teste**
 - Não há nota mínima no TP 😊
 - Não se “descongela” nota TP do ano anterior, salvo exceções indicadas no RAUM
 - Nota **mínima** de 8 no teste ou exame(s)

$$\text{Classificação} = (\text{TP} + \text{teste_ou_exame}) / 2$$



Avaliação – datas importantes

- ?? : Inscrição nos grupos TP do Blackboard
- 7 de Maio: Submissão do TP pelo Blackboard
- 27 a 29 de Maio: Defesa do TP
- 24 de Maio -- Teste
- 14 de Junho -- Exame de recurso



Avaliação

- Prova escrita (teste ou exame) individual e sem consulta.
- É preciso responder **ao problema proposto**;
 - Valoriza-se a capacidade de raciocínio e a concepção de algoritmos (por oposição à utilização de “padrões” de soluções)
 - Quase tudo tem a ver com concorrência
 - As perguntas da parte teórica insistem sempre nos “porquês”, na justificação, demonstração ou prova.

Programa

- Introdução (à *programação de sistemas*)
- Gestão de processos
- Noções de programação concorrente
- Gestão de memória
- Gestão de periféricos
- Gestão de ficheiros

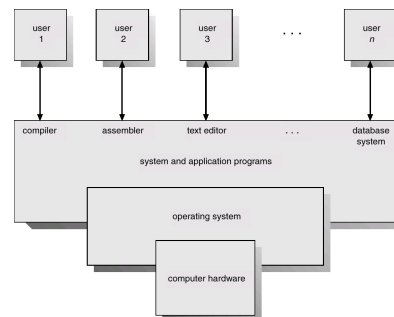
Vamos começar pelo princípio...

Para que serve um computador?

- Para executar programas (aplicações)
- Que facilitam a vida aos utilizadores

O que é um Sistema Operativo?

- Programa que actua como **intermediário** entre os utilizadores e o hardware



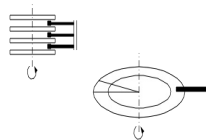
Portanto...

- O SO deve gerir o hardware e colocá-lo à disposição dos programas e utilizadores, de uma forma
 - conveniente,
 - protegida,
 - eficiente,
 - justa,
 - ...



O Sistema Operativo pode ser visto como...

- Extensão da máquina
 - simula uma *máquina virtual* "acima" da máquina real: **open(), read(), write()...**



- Gestor de recursos



Objectivos (1)

- Conveniência
 - SO esconde os detalhes do hardware
e.g. dimensão e organização da memória
 - Simula máquina virtual com valor acrescentado
e.g. cada processo executa numa "máquina"
 - Fornece API mais fácil de usar do que o hardware
e.g. ficheiros vs. blocos em disco



Na prática...




- É o Sistema Operativo quem define a "**personalidade**" de um computador
- Como se comporta o mesmo computador (hardware) após ter arrancado
 - MSDOS?
 - Windows 95?
 - Windows 10?
 - Linux (Ubuntu, Kali...)?



Objectivos (2)

- Eficiência
 - SO controla a alocação de recursos
 - Se 3 programas usarem a impressora ao mesmo tempo → sai lixo?
 - Programa em ciclo infinito → computador bloqueia?
 - Processo corrompe a memória dos outros → programas morrem?
 - Multiplexação:
 - Tempo: cada processo usa o recurso à vez (impressora, CPU)
 - Espaço: recurso é partilhado simultaneamente por vários processos (memória central, disco)

Objectivos (3)

- Recapitulemos então os objectivos gerais de um SO
 - Conveniência
 - Eficiência
- Então, os nossos critérios de avaliação serão...
 -  Dá jeito?
 -  É eficiente ou aumenta a eficiência geral do sistema?
 -  Nem uma nem outra?

Tome nota:

- Este “filme” não é para decorar...
- É para perceber a evolução e os porquês
- Quando terminar, terá ficado a saber
 - os objectivos dos Sistemas Operativos
 - como estes foram sendo atingidos:
 - com muita massa cinzenta
 - e algum apoio do hardware!

No início era assim...

- Acesso livre ao computador
 - Utilizador podia fazer tudo, mas
 - Também tinha de fazer tudo...
- Eficiência era baixa
 - Elevado tempo de preparação
 - Tempo “desperdiçado” com debug

No início era assim...

Exemplo: HP 2114B (1968)



- Comprava-se hardware sem software!
- Dava-se acesso livre ao computador
 - Utilizador podia fazer tudo
(i.e. interagir com o programa)
 - Mas também tinha de fazer tudo...

Sim, **TUDO!!!!**

Altair 8800 (intel 8080)



Sistemas Operativos - 2023/2024

24

Acesso livre

- Utilizador é um faz-tudo (I):
 - Percebe o problema e idealiza a solução (algoritmo + dados)
 - Descreve o algoritmo em “alguma notação de alto nível”
 - Estuda o manual do CPU e memória, faz então a tradução para linguagem máquina (e decide que endereços vai usar)



```
1: S = 0
2: Ler N
3: Se N > 999 continuar no passo 6
4: S = S + N
5: Voltar ao passo 2
6: Mostrar S
```

```
$32, %rsp
$64, %eax
%eax, %edi
$0, -4(%rbp)
callq _malloc
leaq L_str(%rip), %rdi
movl $4294967295, %ecx
xorl %edx, %edx
```

0948	83f3	83ff	1675	8b48	0873
0002	e800	01ae	0000	c085	850f
8b48	105b	09eb	f183	7502	4879
5089	4c00	350b	0312	0000	b141
07eb	f140	48c3	5089	0f00	3b06
86420	8440	78ff	410a	448b	3cbe
86440	4000	0000	55e8	0001	8500
			75c0	0f09	0306

Sistemas Operativos - 2023/2024

25

- A tradução manual é particularmente penosa e facilmente sujeita a erros
- É necessário construir a instrução a partir do “OP code”, flags, registos, endereços, modos de endereçamento...
- [8080 Instruction Set](#)

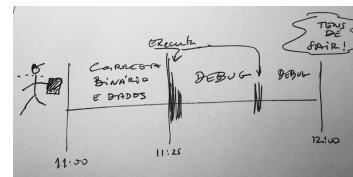
110=P	(S'ign flag not set - POSITIVE)		
111=M	(S'ign flag set - MINUS)		
Inst	Encoding	Flags	Description
MOV D,S	01DDSSSS	-	Move register to register
MVI D,#	00DDD110 db	-	Move immediate to register
LXI RP,#	00RFP0001 1b hb	-	Load register pair immediate
LDA a	00111010 1b hb	-	Load A from memory
STA a	00110010 1b hb	-	Store A to memory
LHLD a	00101010 1b hb	-	Load H:L from memory
SHLD a	00100010 1b hb	-	Store H:L to memory
LDAX RP	00RP1010 *1	-	Load indirect through BC or DE
STAX RP	00RP0010 *1	-	Store indirect through BC or DE
XCHG	11101011	-	Exchange DE and HL content
ADD S	10000SSS	ZSPCA	Add register to A
ADI #	11000110 db	ZSPCA	Add immediate to A

Sistemas Operativos - 2023/2024

26

Acesso livre

- Utilizador é um faz-tudo (II):
 - Chegada a hora, carrega manualmente o programa e dados
 - Executa o programa
 - Se não está correcto, tem de descobrir sozinho os erros
 - Eficiência é muito baixa devido ao desperdício de tempo de CPU



- Carregamento manual demorado
- Debug muito demorado...
 - Erro no algoritmo?
 - Ao traduzir para assembly?
 - Ao traduzir para binário?
 - Ao carregar programa e dados?

Sistemas Operativos - 2023/2024

27

Para aumentar a eficiência (I)

- Surgiu a ideia de reduzir a intervenção humana, fazendo a preparação de programas e dados *off-line* e acelerando o seu carregamento posterior.
- Inventam-se periféricos de *input* e *output*, como a célebre Teletype, embora a velocidade ainda seja muito baixa (10 chars/s) para ler da fita ou escrever na fita/papel
- E aparece um **loader*** residente

(*) Software que entende o que está na fita e carrega para o local pretendido da RAM os bytes que vai lendo



When apps were cards or tape...



Card reader + line printer



Para aumentar a eficiência (II)

- Automatizou-se uma parte do “procedimento”
 - Utilizador deixa de interagir com o seu programa, usa fita perfurada ou coloca os cartões num cesto e espera... horas
 - Operadores recolhem o cesto periodicamente e colocam programas e dados no leitor. O sistema executa os jobs e imprime os resultados, que são devolvidos a determinadas horas
- **Ganhou-se em eficiência, perdeu-se em conveniência**
 - Um job que antes demorava uma hora é agora executado em segundos
 - *Turnaroud* time de horas: entrega às 9, recebe às 19

Melhor do que um operador...

- É ter um *programa* que:
 - Automatize a operação do computador, passando ao job seguinte sempre que o job que detém o cpu chega ao fim
 - Operador apenas carrega / descarrega cartões e junta as listagens aos respectivos cartões, para os utilizadores verem os resultados
- Será o início de uma Job Control Language (JCL) e de um interpretador de comandos ?

Embrião de um sistema operativo?

Para aumentar a eficiência (III) ...

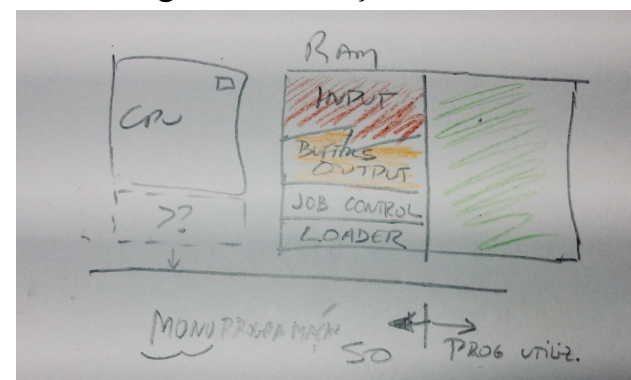
Apesar de cada job “curto” demorar (dezenas de) segundos, o CPU está ocupado todo esse tempo, ora a executar o código ora em espera activa de IO.

E se... input, execução e output pudessem ser realizados em paralelo?

Para aumentar a eficiência (IV)



Como gerir a execução concorrente?

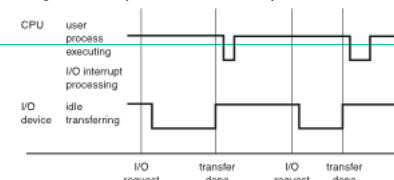


Mas havia o risco de...

- Perder eficiência devido a erros de programação
 - Ciclos infinitos
 - Espera activa por periféricos lentos
 - Erros na leitura ou escrita de periféricos
 - Programa do utilizador destruir o “programa de controle”

Soluções (hardware)

• Interrupções



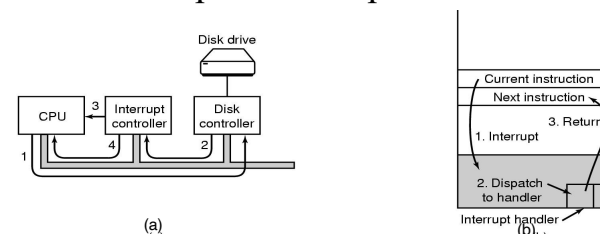
- Relógio de **Tempo Virtual**
- Instruções privilegiadas, 2 ou mais *modos de execução*, protecção de memória
- E aparecem as **system calls** !

Exemplo: Polling IO

- Disk_IO()
 - Carrega o controlador de disco com parâmetros adequados (pista, sector, endereço de memória, direcção...)
 - While (NOT IO_done); /* do nothing in a **busy way***/
- (Equivalente a:
 Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste?
 Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste?
 Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste?
 ...)
- OK, regressa de disk_io()

Resulta em **desperdício de tempo de CPU**

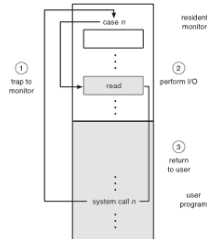
Exemplo: Interrupt-driven IO



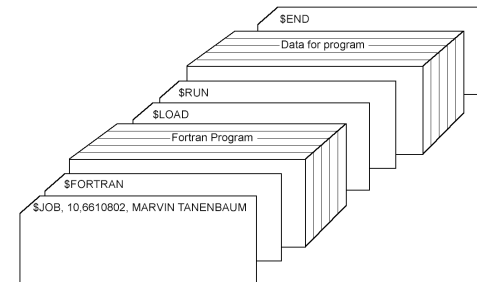
- (a)
- (a) OS inicia operação de IO e prepara-se para receber a interrupção; entretanto pode estar executando outras tarefas,
- (b) No fim da operação de IO, o programa em execução é interrompido momentaneamente, SO trata o evento, e decide se o processo que pediu IO retoma a execução, se mantém o que foi interrompido ou até se muda para outro pronto a executar.

Soluções (software)

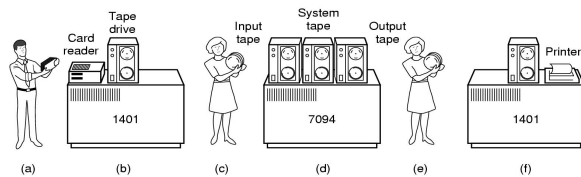
- Chamadas ao Sistema
- Virtualização de periféricos
 - Por exemplo o Leitor de cartões:
 - Programa pede para ler do periférico
 - SO devolve o conteúdo de um cartão que foi copiado para banda magnética ou lido anteriormente para memória (SPOOL)
- Mais tarde vai surgir a **multiprogramação**



Exemplo de um “job”



Primeiros sistemas de batch

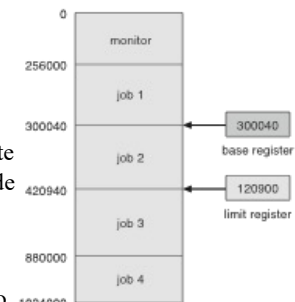


- Processador auxiliar faz IO de periféricos lentos (virtuais)
- Carregar cartões no 1401, que os copia para banda magnética
 - Colocar banda no 7094 e executar os programas
 - Recolher banda com resultados do batch e colocá-la no 1401, que os envia para a impressora
- (Leitor e impressora seriam mais rápidos do que a imagem sugere... Ver slide 30)

Multiprogramação

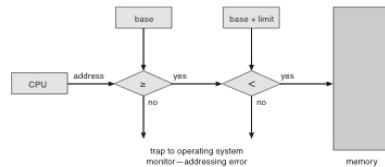
- Memória central dividida em várias zonas (partições)
- Vários jobs simultaneamente em memória mas não “vêm” os outros
- Isto permite executar concorrentemente vários processos, repartindo o tempo de cpu entre eles

Já não obriga a encadear input, execução e IO:





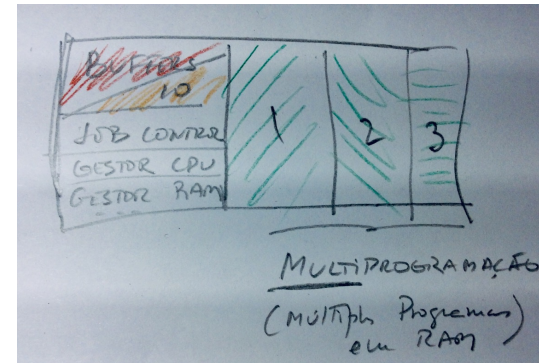
Protecção de memória



Note que estes testes têm de ser feitos sempre que há um acesso à memória...
2, 3 ou mesmo 4 vezes por instrução?



Multiprogramação



E a conveniência?

- Reaparece com os sistemas de

Time-Sharing

- Terminais (consolas) ligados ao computador central permitem que os utilizadores voltem a interagir directamente
- Sistema Operativo reparte o tempo de CPU pelos programas prontos a executar (de preferência carregados em memória)

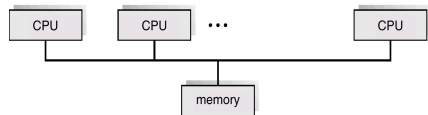


E desde aí?

- Com o computador pessoal volta tudo ao início...
 - Control Program for Microcomputers
 - Monoprogramação, baixa eficiência...
- Mas...
 - É muito conveniente para o utilizador
 - É barato, logo eficiência não é a prioridade



Multiprocessamento (1)



- A ideia é executar mais carga no mesmo intervalo de tempo (i.e. ter **throughput** maior). Não é executar um programa muito mais depressa (i.e. baixar o seu **tempo de resposta**). Para isso é preciso dividir uma aplicação em vários processos e executar cada um em seu CPU/core.
- No entanto, consegue reduzir o tempo de espera até ter acesso a um CPU



Multiprocessamento (2)

• Arquitectura

– Simétrico

- qualquer CPU pode executar código do SO
- cuidado com *race conditions*, (e.g. acesso à tabela de blocos de memória livres)
- hardware mais sofisticado (e.g. disco interrompe todos os CPUs?)

– Assimétrico

- Periféricos associados a um só CPU, o que executa o SO
- CPUs podem estar parados porque o SO não “despacha”



Sistemas Distribuídos (1)

- Nos anos 80 apareceram as redes locais para partilha de
 - recursos caros (e.g. impressoras) ou
 - inconvenientes de replicar (e.g. sistemas de ficheiros)
 - redirecionamento de IO

Exemplo: `cat fich.txt | rsh print_server lpr`

• Questões

- protocolos de comunicação, modelo cliente-servidor?
- como saber o estado de recursos remotos?



Sistemas Distribuídos (2)

• Em breve se passou

- dos *network aware OSs*, que já permitiam acesso remoto a discos, sistemas de ficheiros, impressoras...
- para sistemas vocacionados para o trabalho em rede

• E chegou-se à Web...



E ainda...

- SOs para *mainframes*:
 - IBM MVS, IBM VM/CMS.
 - desenvolvidos nos anos 60 e ainda em operação (z/VM)!
- A caminho dos 60 anos, a virtualiação de ambientes de execução mantém a sua importância: Vmware, Docker...

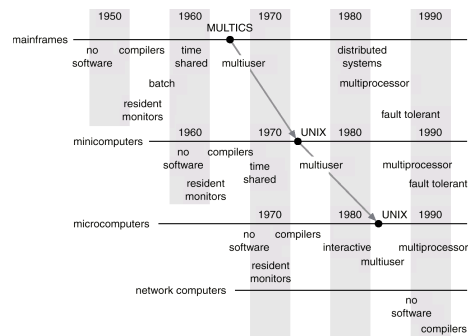
E ainda...

(apesar de não fazerem parte do programa)

- SO de Tempo Real
 - controlo de processos industriais, sistemas de voo, automóveis, máquinas de lavar, etc.
 - SO normais não conseguem dar **garantias** de tempo de resposta.
- SOs para sistemas “restritos”: microcontroladores, redes de sensores



Evolução de conceitos de SO



Antes de continuar...



Assegure-se que percebeu os conceitos anteriores, e que entendeu os problemas que as soluções indicadas procuram resolver...

- Sabe o que são e para que servem os 2 modos de execução?
- Modo de execução é **hardware** ou **software**?
- E multiprogramação? E multiprocessamento?
- E interrupção? Para que servem as interrupções?
- Para que servem as system calls? Qual a diferença em relação a uma função normal?
- O que é o tempo virtual?



Recorde ainda....

- CPU
 - Registos (PC, SP, BP, CS, DS...) → “contexto volátil”
 - Instruções privilegiadas → só podem ser executadas em modo “protegido”; a forma de um programa do utilizador solicitar serviços ao SO é através das chamadas ao sistema (syscalls)
 - Interrupções (já agora, recordemos **traps** e **exceções**!)
- Memória (mas o que é um endereço? E modos de endereçamento?)
- Periféricos + formas de dialogar com eles



Programas versus processos

- Programa executável:
 - Resultado da compilação, ligação, (re)colocação em memória
 - Normalmente dependerá de módulos externos, libs
 - Também pode ser um script interpretado (pela bash?)
- Processo em execução:
 - código já (re)colocado em memória central + dados + stack
 - Estruturas de gestão:
 - Processo: contexto, recursos HW e SO em uso (registos, ficheiros abertos...)
 - Utilizador (uid, gid, account...)



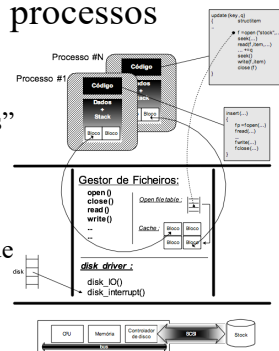
Competição entre processos

- Como surgem as “race conditions”
 - entre processos
 - dentro do SO

- Vantagens/desvantagens do uso de caches



- Note que estamos a falar de caches por software, de cópias de dados em memória mas acessíveis em contextos diferentes



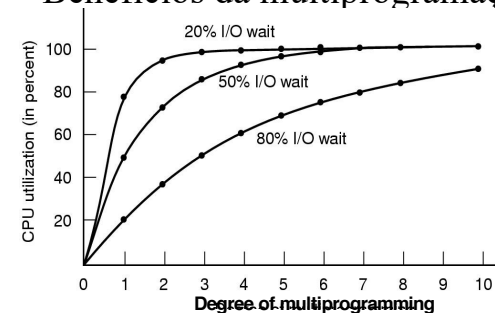
Programa

- Introdução
- Gestão de processos
- Noções de programação concorrente
- Gestão de memória
- Gestão de periféricos
- Gestão de ficheiros

Porquê criar vários processos?

- Porque dá jeito... **+ conveniência**
 - Estruturação dos programas
 - Para não estar à espera (spooling, background...)
 - Multiplas actividades / janelas
- Porque é melhor **+ eficiência**
 - Múltiplos CPUs
 - Aumenta a utilização de recursos (e.g multiprogramação)

Benefícios da multiprogramação



Processos

- Processo: um programa em execução, tem actividade própria
- **Programa**: entidade *estática*, **Processo**: entidade *dinâmica*
- Duas invocações do mesmo programa resultam em dois processos diferentes (e.g. vários utilizadores a usarem cada um a sua shell, o vi, browser, etc.)

Processos

- O contexto de execução de um processo (i.e. o seu **estado**) compreende:
 - código
 - dados (variáveis globais, *heap*, *stack*)
 - estado do processador (registos)
 - ficheiros abertos,
 - tempo de CPU consumido, ...



Exemplo de informação sobre um processo

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		



Gestão de Processos

- Os programas em execução requerem tempo de CPU, memória, utilização de dispositivos, entradas livres em estruturas de dados do SO, canais de comunicação com outros processos, etc.:
 - COMPETEM POR RECURSOS (hw e sw)
- A competição é no entanto mediada pelo SO. Como os processos não podem aceder directamente ao código e dados do SO, necessitam de uma API para solicitar serviços através das *system calls*



Exemplo: (parte da) API de processos em Unix

- Para criar um novo processo:
 - fork**: cria um novo processo (a chamada ao sistema retorna “duas vezes”, uma para o pai e outra para o filho)
 - A partir daqui, ambos executam o mesmo programa
- Para executar outro programa dentro do mesmo processo
 - exec**: substitui o programa por um novo programa
- Para terminar a execução
 - exit**

Compare o **exec** com a invocação de uma função: são muito diferentes



fork/exec

```
pid = fork()
if (pid == 0) { // Sou o filho
    exec( novo programa )
} else {
    // Código do pai
}
```

Gestão de Processos

Cabe ao sistema operativo fazer o escalonamento dos processos, garantindo que a ordem de acesso ao CPU correspondente às políticas de escalonamento previamente definidas

É preciso definir **OBJECTIVOS**

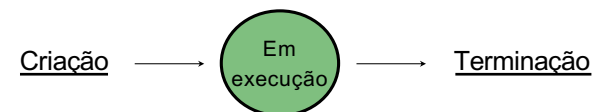
Objectivos

- Conveniência:
 - Justiça
 - Redução dos tempos de resposta
 - Previsibilidade
- Eficiência:
 - Débito (*throughput*), transacções por segundo ...
 - Maximização da utilização de CPU e outros recursos
 - Favorecer processos “bem comportados”, etc.

Critérios de escalonamento

- IO-bound ou CPU-bound
- Interactivo ou não (batch, background)
- Urgência de resposta (e.g. tempo real)
- Comportamento recente (utilização de memória, CPU)
- Necessidade de periféricos especiais
- PAGOU para ir à frente dos outros...

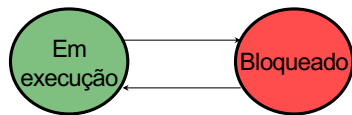
Estados de um processo (i)





Estados de um processo (ii)

Podemos para já admitir que durante a sua “vida” os processos passam por 2 estados:

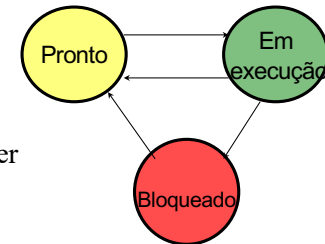


Estados de um processo (iii)

Na prática, há mais processos não bloqueados do que CPUs

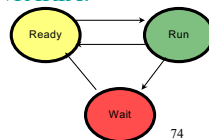
Surge uma fila de espera com processos **Prontos a executar**

Processos em execução podem ser desactivados

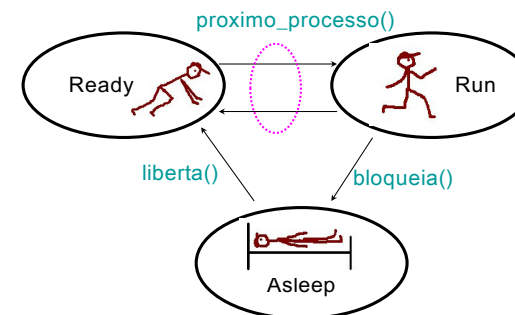


Estados de um processo (iv)

- **Em execução**
 - Foi-lhe atribuído o/um CPU, executa o programa correspondente
- **Bloqueado**
 - O processo está logicamente impedido de prosseguir porque lhe falta um recurso ou espera por evento
 - Do ponto de vista do SO, é uma transição **VOLUNTÁRIA!**
- **Pronto a executar**, aguarda escalonamento



Primitivas de despacho (i)



Primitivas de despacho (ii)

- Bloqueia(evento)
 - Coloca **processo corrente** na fila de processos **parados** à espera deste “evento”
 - Invoca próximo_processo()
- Liberta(evento) ou liberta(processo,evento)
 - Se o **outro** processo não está à espera de mais nenhum evento, então coloca-o na lista de processos **prontos a executar**
 - Nesta altura pode invocar ou não próximo_processo()



Primitivas de despacho (iii)

- Proximo_processo()
 - Selecciona um dos processos existentes na lista de processos prontos a executar, de acordo com a política de escalonamento
 - Executa a comutação de contexto
 - Salva contexto volátil do processo corrente
 - Carrega contexto do processo escolhido e regressa (executa o return)

Como o Stack Pointer foi mudado, “regressa” para o **processo escolhido!**

Principais decisões

- Qual o próximo processo?
- Quando começa a executar?
- Durante quanto tempo?
- Por outras palavras,

Há **desafecção forçada** ou não?

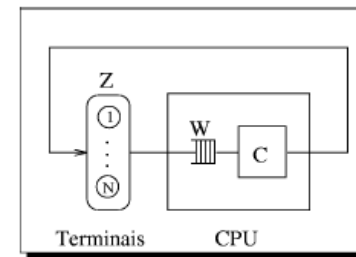
Escalonamento de processos

- Quando, uma vez atribuído a um processo, o CPU nunca lhe é retirado então diz-se que o escalonamento é **cooperativo** (non-preemptive).
 - Exemplos: Windows 3.1, co-rotinas, thread_yield()
- Quando o CPU pode ser retirado a um processo ao fim do quantum ou porque surgiu outro de maior prioridade diz-se que o escalonamento é com **desafecção forçada** (preemptive)

Escalonamento de processos

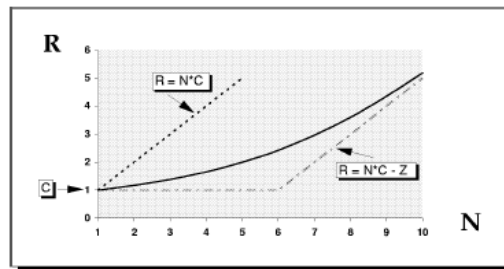
- Escalonamento **cooperativo** (non-preemptive).
 - “poor man’s approach to multitasking” ?
 - Sensível às variações de carga
- Escalonamento com **desafectação forçada**
 - Sistema “responde” melhor
 - Mas a comutação de contexto tem overhead

Modelo de sistema interactivo

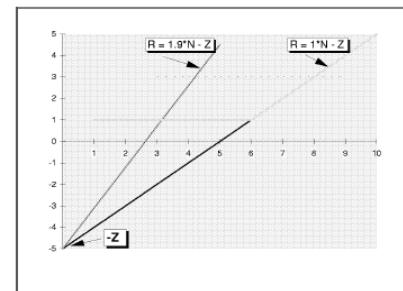


Z = Think time
C = Service time
W = Wait time
N = Number of users

Tempo de Resposta (carga homogénea)



Tempo de Resposta (carga heterogénea)



Assuma-se agora que uma em cada 10 interações é muito longa, 10 vezes maior. Veja-se a degradação de tempos de resposta

Tempo de Resposta (carga heterogénea)

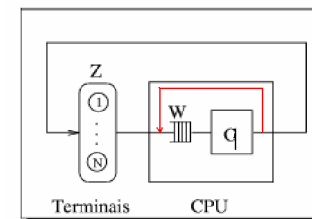
- Para evitar que as interações longas monopolizem o CPU e aumentem o tempo de resposta das restantes deve usar-se desafectação forçada.
- Neste caso deve atribuir-se um quantum (ou time slice) para permitir a troca rápida de processos:
 - Interações curtas terminam dentro dessa fatia de tempo, logo não são afectadas pela política de desafectação.
 - Interações longas executam durante um quantum e a seguir o processo correspondente regressa ao estado de **Pronto a Executar**, dando a vez a outros processos. Mais tarde ser-lhe-á atribuído nova fatia de tempo, e sucessivamente até a interação terminar.

Duração da fatia de tempo

- Maioria das interacções deve “caber” num quantum
- Se precisar de 2 passagens pelo CPU, T_{Resposta} é quase o dobro!

$$R = W + C$$

$$R = W + q + W + c'$$



Escalonamento de processos

- Escalonadores de longo-prazo (segundos, minutos) e de curto-prazo (milissegundos)
- Processo CPU-bound: processo que faz pouco I/O mas que requer muito processamento
- Processo I/O-bound: processo que está frequentemente à espera de I/O.

Escalonamento de processos

- Os processos prontos são seriados numa fila (*ready list*)
- A lista é uma lista ligada de apontadores para PCB's
- A lista poderá estar ordenada por vários critérios de forma a dar tratamento preferencial a alguns processos

Nesta UC deve evitar falar em *prioridade* sem descrever e justificar o critério que leva a uma ordenação particular.

Escalonamento de processos

- Quando um processo é escalonado, é retirado da *ready list* e posto a executar
- Pode “perder” o CPU por 2 razões:
 - Faz um pedido de I/O que não pode ser servido imediatamente (e.g. teclado) ou pede ao SO para esperar: passa ao estado de **bloqueado**
 - É *desafectado* porque aparece um processo com maior "prioridade" ou o seu *quantum* expira: passa ao estado de **pronto**

Escalonamento de processos

- Pretende-se maximizar a utilização do CPU tendo em atenção outros aspectos:
 - Tempo de resposta para aplicações interactivas
 - Utilização de dispositivos de I/O
 - Justiça na distribuição do tempo de CPU

Escalonamento de processos

- A decisão de escalonar um processo pode ser tomada em diversas alturas:
 - Quando o processo é bloqueado (óbvio!)
 - Quando o processo passa a pronto a executar
 - Quando se completa uma operação de I/O
 - Quando um processo termina

Escalonamento de processos

- Diferentes algoritmos de escalonamento visam objectivos diferentes:
 - Diminuir o tempo de resposta (reduzindo o tempo de espera para determinados processos)
 - Máximizar a utilização do CPU

Escalonamento de processos

- Alguns algoritmos de escalonamento:
 - FCFS (First Come, First Served)
 - SJF (Shortest Job First)
 - SRTF (Shortest Remaining Time First)
 - Preemptive Priority Scheduling
 - RR (Round Robin)

First Come, First Served (FCFS)

- A *ready list* é uma fila FIFO
- Os processos são colocados no fim da fila e selecionado o da frente
- Método cooperativo
- Nada apropriado para ambientes interactivos

FCFS

- Uma vantagem óbvia do FCFS é sua simplicidade de implementação: lista de processos por ordem de criação do processo (batch)
- Sujeito a tempos de espera com grandes flutuações, dependendo da ordem de chegada e das características dos processos: “efeito de comboio”
- Parece haver vantagens em escalonar os processos mais curtos à frente...

SJF (Shortest Job First)

- A ideia é escalonar o processo mais curto primeiro
- Possibilidades:
 - Desafectação forçada (SRTF) – interrompe-se o processo em execução se aparecer um mais curto
 - Cooperativo – mesmo na presença de um processo mais curto, pode aguardar-se pela terminação ou bloqueio voluntário do processo em execução. E nessa altura escolhe-se o mais curto

Preemptive Priority

- Associa uma prioridade (geralmente um inteiro) a cada processo.
- A *ready queue* é uma fila seriada por prioridades.
- Escalona sempre o processo na frente da fila.
- Se aparece um processo com maior prioridade do que o que está a executar faz a troca dos processos

Preemptive Priority

- Problema: starvation
- Uma solução: envelhecimento – aumenta a prioridade dos processos pouco a pouco de forma a que inevitavelmente executem e terminem.
- Convém justificar quando e quanto aumenta...

RR (Round Robin)

- Dá a cada processo um intervalo de tempo fixo de CPU de cada vez
- Quando um processo esgota o seu quantum retira-o do CPU e volta a colocá-lo no fim da fila.
- Ignorando os overheads do escalonamento, cada um dos n processos CPU-bound terá $(1/n)$ do tempo disponível de CPU

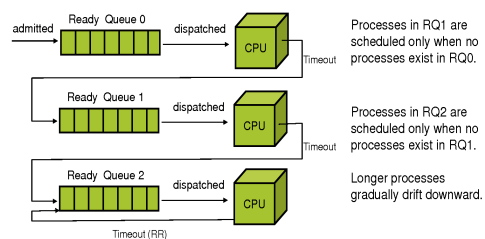
RR

- Se o quantum for (muito) grande o RR tende a comportar-se como o FCFS
- Se o quantum for (muito) pequeno então o overhead de mudanças de contexto tende a dominar degradando os níveis de utilização útil de CPU
- Favorece processos que libertem o CPU ao fim de pouco tempo“ (aproxima-se do “shortest”) mas sem exigir conhecimento rigoroso do tempo de cada “CPU burst”



Multilevel Feedback Queue Scheduling

- Another way to put a preference on short-lived processes
 - Penalize processes that have been running longer.
- Preemptive



Níveis de escalonamento

- Uma vez que há inúmeros critérios de escalonamento e muitas variáveis a considerar para saber qual o “melhor” processo a escolher, é habitual dividir a questão em 2 ou 3 níveis:
 - Nível 0 --- só despacha o que está em RAM (RR ou MLQ)
 - Nível 1 --- Decide que processos são multiprogramados, por indicação do gestor de memória
 - Nível 2 --- Não deixa criar processos nas horas de ponta, por decisão se quem administra e conhece a carga diária.