Eduardo Santos 14/01/2020

Resumos exame teórico SO

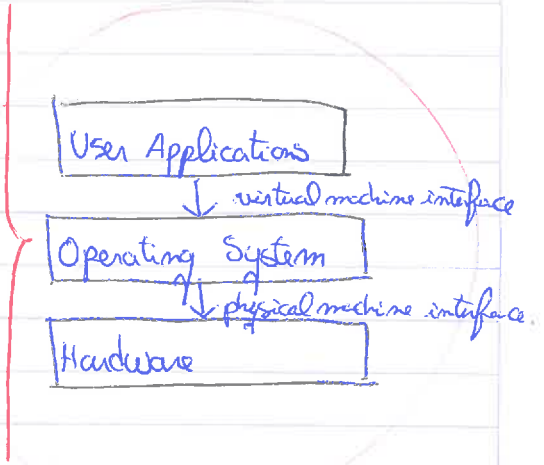
Sistema Operativo

↳ programa base que estabelece a interface entre os programas de aplicação e o hardware.

Hardware → Physical devices
↳ Microarchitecture
↳ Machine language

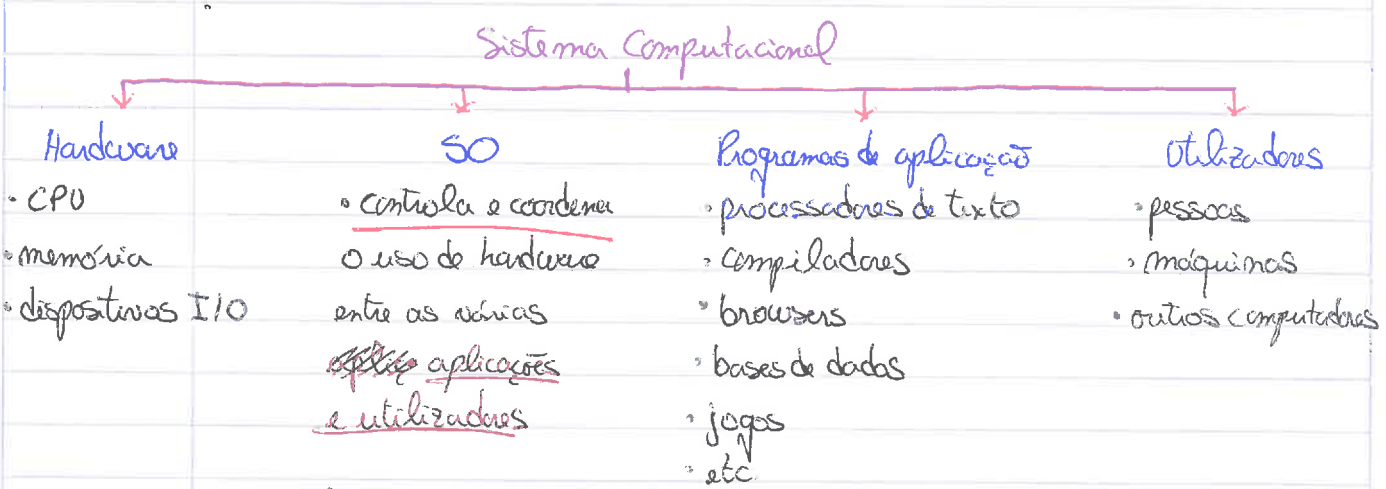
System Programs → ~~Operating~~ Operating System
↳ compilers
↳ Editors
↳ command interpreter

Application Programs



Objetivos do Sistema Operativo:
→ executar os programas de aplicação
→ tornar o hardware + fácil de usar
→ usar o hardware de forma eficiente, gerindo os recursos do mesmo.

↳ ligação direta entre OS e hardware



Sistema Operativo fornece:

- Serviços → o SO cria serviços standard que são implementados pelo hardware.
  ↳ Exemplos: sistema de ficheiros, memória virtual, redes, etc.
  ↳ SO como criador de uma máquina virtual.

- Coordenação → o SO coordena as várias aplicações e utilizadores de modo a garantir segurança, eficiência e justiça na utilização dos recursos.
  ↳ exemplos: concorrência, proteção da memória, segurança.
  ↳ SO como gestor de recursos.

- controlo ↔ O SO controla a execução dos programas prevenindo erros e uso impróprio do computador.
  - ↳ exemplos: escalonamento do CPU, criação de novos processos, seg fault, etc. ↳ falado + à frente

O objetivo é criar um SO que é simultaneamente fácil de usar e eficiente.

Papéis do SO

- ↳ Árbitro: • gere recursos partilhados: CPU, memória, discos, impressoras, etc.
- ↳ Ilusionista: • fornece às aplicações/programador abstrações de recursos com capacidades superiores às existentes: memória infinita; uso exclusivo do CPU, etc.
- ↳ Adaptador: • serviços comuns: sistema de ficheiros; rotinas da UI.
  • separa aplicações dos dispositivos de entrada/saída.

Funcionalidades criadas:

→ estabelecimento do ambiente de base de interação com o utilizador.
→ mecanismos de execução controlada de programas.
→ mecanismos de comunicação entre programas e respetiva sincronização.
→ disponibilização de facilidades pra o desenvolvimento, teste e depuração de programas.
→ espaço de endereçamento virtual dos programas é independente dos limitações da memória física.
→ sistemas de ficheiros
→ modelo geral de acesso a dispositivos de I/O.
→ deteção de situações de erro.
↳ falado/explicado + à frente

Funcionalidades

→ Concorrência: • permite que vários programas sejam executados em simultâneo
• tambem vários utilizadores em simultâneo. ↳ escalonamento

→ Dispositivos de I/O: • CPU continua a trabalhar enquanto I/O não responde.
• mecanismos comuns para acesso a vários tipos de dispositivos.

→ Gestão da memória: • SO gere as alocações de memória e transferências de dados entre memória e disco.

→ Ficheiros: • espaço em disco é organizado num sistema de ficheiros capaz de armazenar vários ficheiros de tamanho variável.

-> Sistemas distribuídos e redes: • permite que um grupo de computadores trabalhem de forma conjunta para resolver um problema.
↳ próximo semestre
nice

Organização do computador
-> CPUs e controladores de dispositivos de I/O executam em paralelo.
-> cada controlador de dispositivo trata um tipo particular.
-> controladores de dispositivo têm buffer local.
-> CPU move dados de/para memória e de/para buffers locais
-> transferências de I/O são do dispositivo para o buffer local do respetivo controlador e depois para a memória.
-> controlador do dispositivo informa CPU que terminou a operação através a operação através do envio de uma interrupção.

System Calls

Modos de operação
-> de modo a garantir a segurança do sistema, a maioria dos SOs podem executar em 2 modos:
• modo de utilizador
↳ Com restrições de segurança
↳ acesso a certas instruções e zonas de memória e dispositivos estão interditos.

• modo do kernel
↳ sem restrições de segurança
↳ pode executar todas as instruções e acessos
↳ instruções priviligiadas.

chamadas ao sistema providenciam uma forma segura de alternar entre os 2 modos.

chamadas ao sistema
-> interface para acesso aos serviços do SO
-> tipicamente escrita numa linguagem de alto nível
-> programas usam, em geral, uma API para acesso a system calls em vez da utilização direta.

-> as 3 APIs mais comuns são:

- Win32 API para Windows
- POSIX API para sistemas baseados em POSIX (UNIXs, MAC OS X)
- Java API para a Java Virtual Machine.

UNIX	Win32	Description
fork		create a new process
waitpid		can wait for a process to exit
execve		Create Process = fork + execve
exit		terminate execution
open		create a file or open an existing one
close		bla bla bla
read		bla bla file
write		bla bla file
lseek		move the file pointer
stat		get varias file attributes
mkdir		create a new directory
rmdir		remove an empty directory
link		create a new link (directory entry) for the existing file
unlink		bla bla bla.
mount		mounts a storage device or filesystem.
unmount		bla bla bla.
chdir		change the current working directory
chmod		change users permitions
kill		send a signal to a process (terminate process)
time		bla bla bla

process management

call	description
pid = fork()	create a child process identical to the parent
pid = waitpid (pid, &statloc, options)	wait for a child to terminate
s = execve (name, argv, environp)	
exit(status)	

file management

call	description
fd = open(file, how, ...)	open a file for reading, writing or both
s = close(fd)	close an open file
n = read(fd, buffer, nbytes)	read data from a file into a buffer
n = write(fd, buffer, nbytes)	write data from a buffer into a file
position = lseek(fd, offset, whence)	
s = stat(name, &buf)	

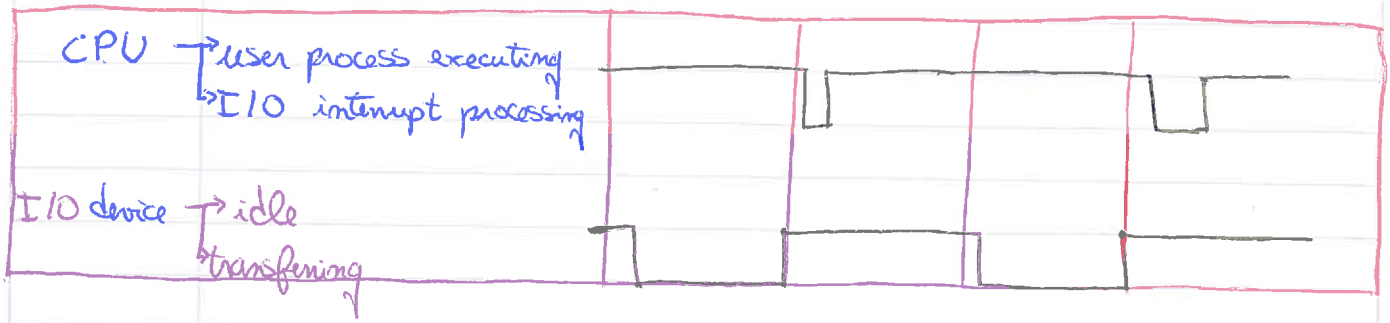
miscellaneous

call	description
s = chdir(dirname)	change the working directory
s = chmod(name, mode)	change a file's protection bits
s = kill(pid, signal)	send a signal to a process

Interrupções / Exceções

- quando o CPU deteta uma interrupção abandona o código que está a executar e transfere o controlo para a rotina de atendimento da interrupção.
- o endereço da instrução interrompida deve ser salvaguardado
- durante a execução da rotina de atendimento as interrupções estão desativadas
  → problema da interrupção perdida
- um trap ou exceção é uma interrupção gerada por software
  → access violation, breakpoint, misaligned access, divide by 0, overflow, illegal instruction, privileged instruction.
- nos SOs as interrupções são fundamentais

Atendimento de uma interrupção



- ~~dispositivo~~ dispositivo de I/O envia interrupção ao CPU quando "transfer done"
- CPU executa rotina de atendimento à interrupção, no âmbito do SO, e volta a executar processo do utilizador.

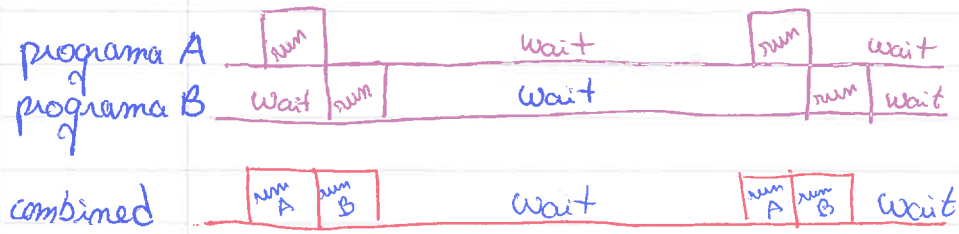
Organização de I/O

registers
↑ ↓
cache
↑ ↓
main memory
↑ ↓
electronic disk
↑ ↓
magnetic disk
↑ ↓
optical disk
↑ ↓
magnetic tapes

Multiprogramação

- permite que + do que 1 processo (programa em execução) esteja ativo em simultâneo.
- a utilização de apenas 1 processo não permite manter o CPU constantemente ativo
- multiprogramação escolhe um novo processo para ocupar o CPU quando o processo que estava a ser executado tem de esperar (ex: chamada de I/O).

- os processos ativos são mantidos em memória
- escalonamento de processos



Timesharing

- CPU altera o processo em execução mesmo que este não necessite de esperar.
- a cada processo é atribuído um tempo máximo de ocupação consecutiva do processador.
- se esse tempo é esgotado o SO muda o contexto do processador para outro processo.
- diminui muito o tempo de resposta de aplicações.
- permite que vários utilizadores usem o mesmo sistema computacional como se dispusessem do sistema em exclusivo.

Projeto do SO

mecanismos → como realizar a função
política → o que vai acontecer

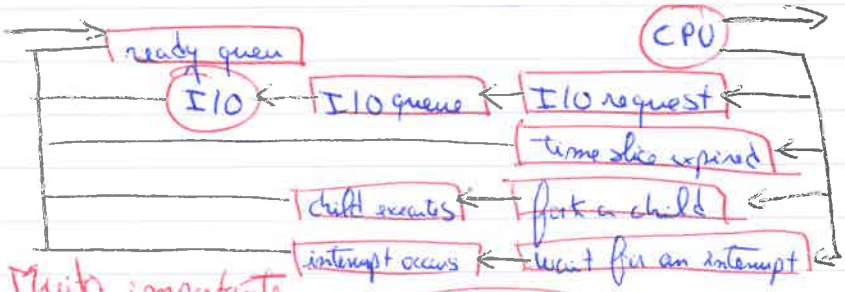
- conceitos a separar:
  - → política - o que será realizado?
  - → mecanismo - como será realizado?
- mecanismos determinam como realizar a função, enquanto que a política define o que vai acontecer
  - → a separação permite aumentar a capacidade de adaptação do sistema.
  - → ao integrar mecanismos sem política associada, o sistema torna-se facilmente adaptável a diferentes políticas.

- monolítico
  - → SO contem todas as funcionalidades de forma estática
  - → permite código otimizado, pouco flexível, ocupa mais memória.
  - → ex.: MS-DOS, UNIX

- modular
  - → sistema operativo permite adição/configuração de funcionalidades através de integração de módulos.
  - → custo/overhead da API, + flexível, menos memória
  - → ex.: Solaris, linux

- microKernel
  - → Kernel apenas com serviços básicos: thread, address space

Processos

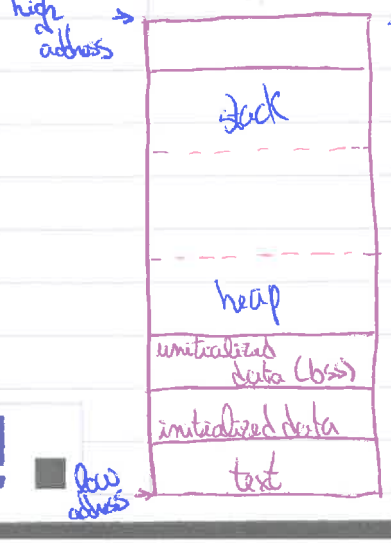


Muito importante

- programa em execução
- cria um processo
  - → inicialização do sistema
  - → execução de chamada ao sistema por processo em execução.
  - → pedido do utilizador para criar novo processo
  - → início de um batch script
- processos podem correr em:
  - → foreground: interage com o utilizador
  - → background: executa sem interação, ~~fork~~ daemon

saber distinguir → simples

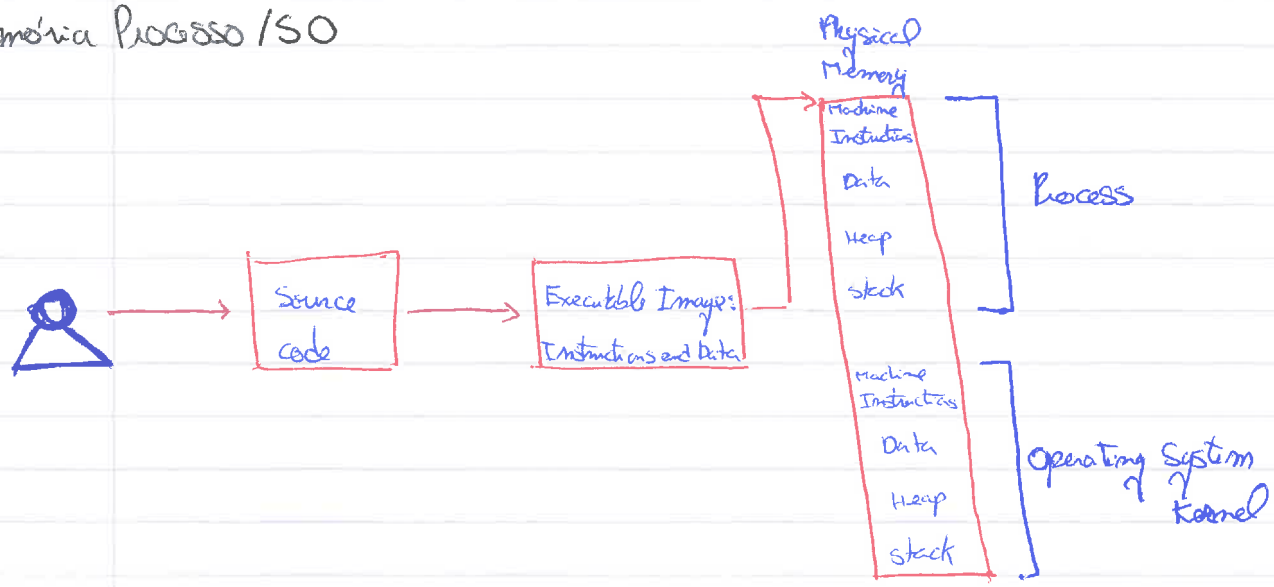
Memória de um processo



↕ command-line arguments and environment variables

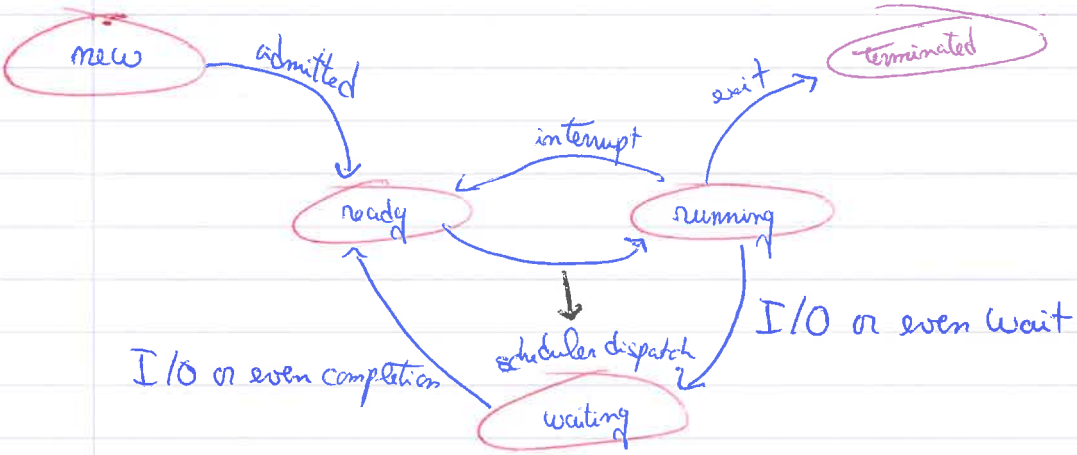
↕ initialized to zero by exec.
read ~~from~~ read from program file by exec.

Memória Processo / SO



Estados de um processo

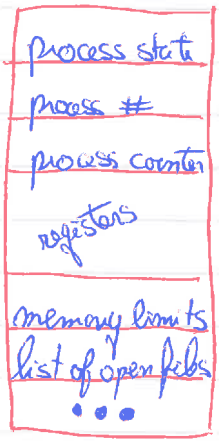
- um processo em execução pode estar nos seguintes estados:
  - → new
  - → running
  - → waiting
  - → ready
  - → terminated



Process Control Block

- o process Control Block (PCB) é a estrutura que, no SO, armazena a info sobre um processo

- inclui os seguintes estados
  - → estado do processo
  - → program counter (PC)
  - → registos do CPU
  - → tipo de escalonamento
  - → info sobre a memória do processo
  - → info sobre o I/O
  - → accounting

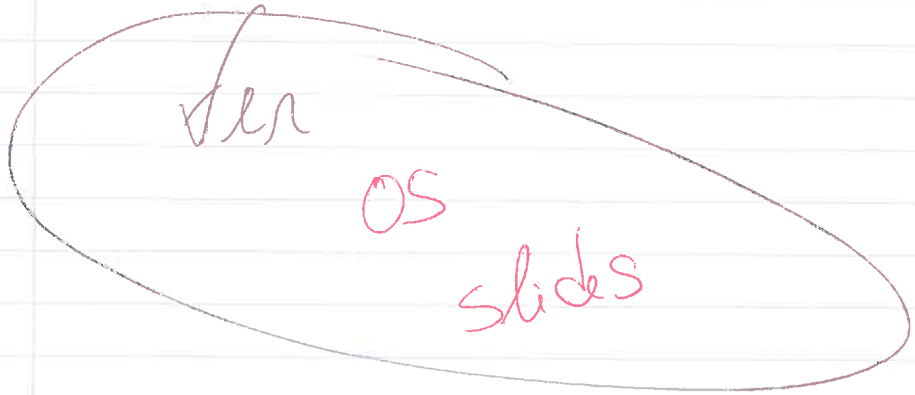


process management
- registos
- PC
- program status word
- stack pointer
- process state
- priority
- scheduling parameters
- process ID
- parent process
- process group
- signals
- time when process started
- CPU time used
- children's CPU time
- time of next alarm

memory management
- pointer to text segment
- " " data "
- " " stack "

resumidamente: ponteiros

file management
- root directory
- working "
- file descriptors
- user ID
- group ID



Árvore de Processos

- quando um processo cria um novo processo → por ex.: fork()
  - → processo criador é designado de processo pai
  - → novo processo é designado de processo filho
- pode ser formada uma hierarquia de processos
- hierarquia de processos
  - → processo pode saber pid do pai
  - → quando o filho morre é enviado o sinal SIGCHLD ao pai
  - → pai recolhe exit cod dos filhos
  - → quando o pai morre, o filho é herdado pelo processo 1 (init)
    - ⇒ a partir do kernel 3.4, um processo pode nomear-se como pai dos processos orfãos seus descendentes (ex: systemd, upstart).

Ver os slides

Tipos de processos

- I/O intensivos
  - → fazem muitas chamadas ao sistema relacionadas com I/O
  - → muitos pequenos periodos de utilização do CPU
- CPU intensivos
  - → fazem poucas chamadas I/O
  - → poucos e longos periodos de utilização do CPU
- num sist. com timesharing e de modo a otimizar a utilização do CPU é positivo que a lista de processos em execução seja equilibrada entre os 2 tipos.

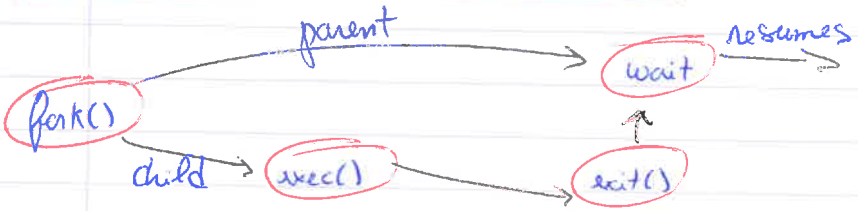
Criação de Processos

- um processo pode criar novos processos
  - → o processo criador designa-se de processo pai e os criados de processos filhos.
  - → os filhos podem, por sua vez, criar novos processos.

- partilha de recursos
  - → pai e filhos partilham recursos
  - → filhos partilham um subconjunto de recursos do pai
  - → pai e filhos não partilham recursos

} ≠s

- execução
  - → pai e filhos executam em paralelo.
  - → pai espera que o(s) filho(s) termine(m).



Criação de processos - POSIX

```
#include <errno.h>
#include <stdio.h>
  ...
pid_t childpid;
  ...
childpid = fork();
switch(childpid)
{
    case -1:
        fprintf(stderr, "ERROR: %s\n", sys_errlist[errno]);
        exit(1);
        break;
    case 0: /* child's code goes here */
        execlp("/bin/ls", "ls", NULL);
        break;
    case default: /* parent's code goes here */
        wait(NULL);
        printf("child completed");
        break;
```

Criação de processos – Win32 e Java

Ver código nos slides

POSIX Input/Output

- abrir e fechar ficheiros

```
int open( const char *path, int oflag, ... /*, mode_t mode */);
int close (int filedes);
```

- ler/escrever

```
ssize_t read (int fd, void *buf, size_t count);
ssize_t write (int fd, const void *buf, size_t nbytes);
```

- duplicar file descriptors

```
int dup (int oldfd);
int dup2 (int oldfd, int newfd);
```

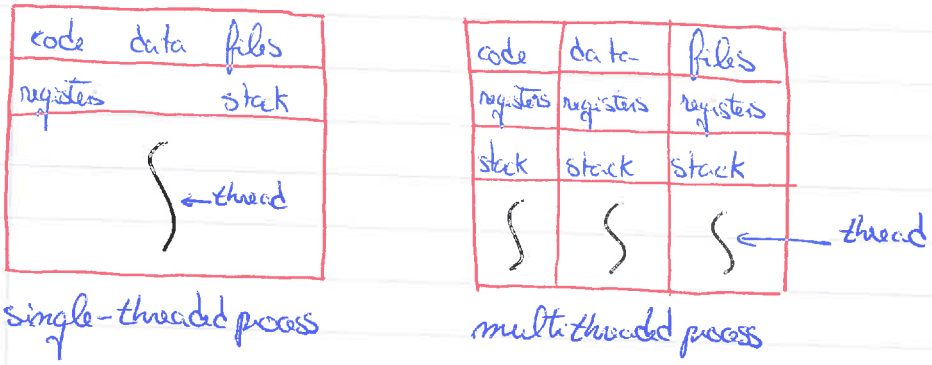
Escrita

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int main (int argc, char *argv[]) {
  int fd;
  fd = open("write1.txt", O_WRONLY...);
  write (fd," message1\n",9);
  close(fd);
  return 0;
}
```

Redir output no processo filho

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    int fd
    switch(fork()){
        case 0: //child
            fd = open ("redirforkexec1.txt","wb");
            dup2(fd,1); // close 1, then make 1 refer to same file as fd
            close(fd);
            execlp ("ls","ls",NULL); // exec ls
            break;
        default: //parent
            printf ("pid = %d\n",getpid());
            break;
    }
    printf("END.\n");
    return 0;
}
```

Processo Single e Multi threaded



single-threaded process

multithreaded process

(I guess)

Threads

Processos e Threads

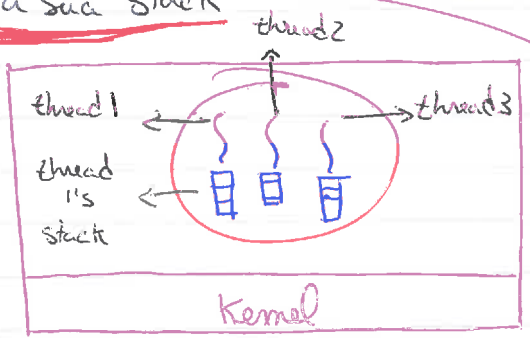
- cada processo tem:
  - → address space
  - → global variables
  - → open files
  - → child processes
  - → pending alarms
  - → signals and signal handlers
  - → accounting information

- cada thread tem:
  - → PC (program counter)
  - → Registers
  - → stack
  - → state

O que é uma thread?

- O kernel distribui o tempo do CPU para cada programa, ao nível do milisegundo. Assim, dá a impressão que 2 ou mais programas correm em paralelo. Em CPUs com múltiplos núcleos de facto há programas que correm em paralelo.
- Processo é um programa em execução
- Processo pode conter uma ou + threads.
- Thread → conjunto de valores independentes para os registos do processador. como inclui o program counter, cada thread controla a ordem de execução do processo.
- Threads são uma maneira de um programa se dividir em tarefas a executar.

Cada thread tem a sua stack



linha de execução de comandos

Vantagens das threads

- estrutura do programa / Modularidade
- responsividade
- partilha de recursos
- melhor desempenho
- utilização de arquiteturas multi-processador

Diferença entre threads e processos?

- threads são leves, processos são pesados (comparativamente)
- multithreading + eficiente que multiprocessing

Suporte à implementação

- user threads
  - → gestão das threads é realizada por uma biblioteca que corre em modo de utilizador

- Kernel threads
  - → gestão das threads é realizada diretamente pelo kernel
  - → Windows XP/2000, Solaris, linux, Tru64 UNIX, Mac OS X

Modelos Multithreading

- Many-to-one
  - → várias threads do utilizador mapeadas numa thread do kernel
  - → exemplos: ⇉ Solaris Green threads
    - ⇉ GNU Portable threads
  - → se uma thread bloqueia todas bloqueiam
  - → não tira partido de vários processadores
  - ⇉
- One-to-one
  - → cada thread do utilizador mapeada numa thread do kernel
  - → exemplos: ⇉ Windows NT/XP/2000
    - ⇉ linux
    - ⇉ Solaris 9 e post.
  - → nº total de threads do sistema pode ser limitado.
  - → funciona melhor que o modelo many-to-one porque este permite que outra thread corra quando uma faz um blocking system call.
- Many-to-many
  - → várias threads do utilizador mapeadas em várias threads do kernel
  - → exemplos: ⇉ Solaris antes do 9.
    - ⇉ Windows NT/2000 com ThreadFiber
    - ⇉ # de threads do kernel pode variar com aplicação e com sistema

Pthreads → posix threads

- POSIX standard para a criação e sincronização de threads
- API define comportamento, mas não implementação
- comum em sistemas UNIX (Linux, Mac OS X)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	release the CPU to let another thread run
Pthread_attr_init	create and initialize a thread's attribute structure
Pthread_attr_destroy	remove a thread's attribute structure

Criar POSIX threads

- `int pthread_create (pthread_t * thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg);`

- implementação:

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
void *PrintMsg (void * threadid){
    long tid;
    tid = (long) threadid;
    printf ("Hello World! thread ID, %d\n", tid);
    pthread_exit (NULL);
}
int main (int argc, char * argv[]){
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;
    for (i=0; i<NUM_THREADS; i++){
        printf ("main(): creating thread, %d\n", i);
        rc = pthread_create (&threads[i], NULL, PrintMsg, (void *) i);
        if (rc){
            printf ("ERROR: unable to create thread, %d\n", rc);
            exit(1);
        }
    }
    pthread_exit(NULL);
```

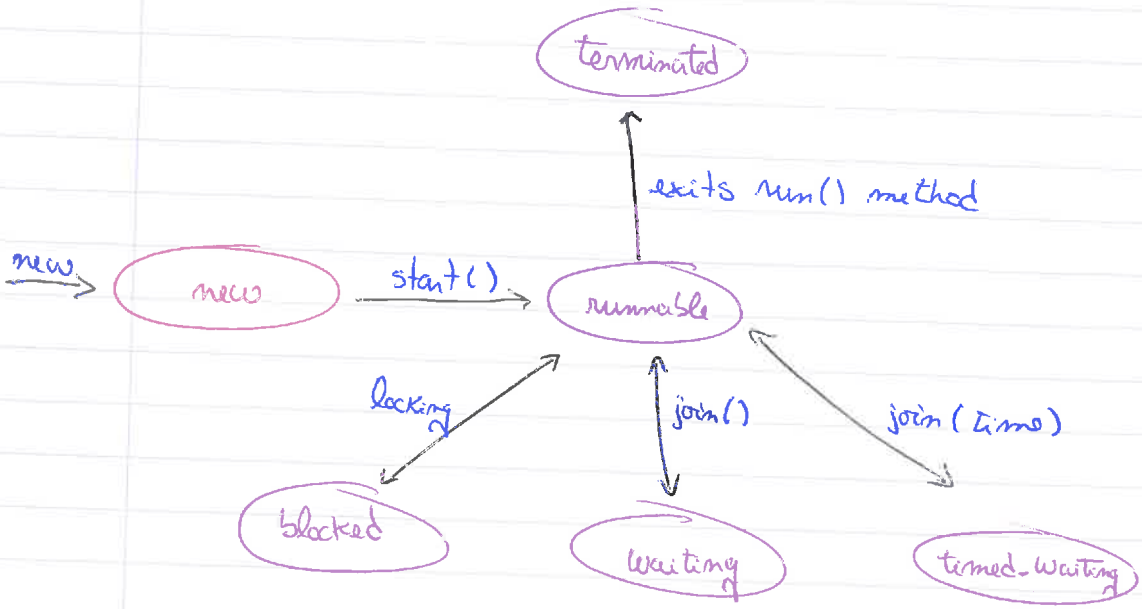
Criar Java thread

Ver código nos slides

Java threads - estados

- **New**
  → thread foi criada mas start() ainda não foi chamado
- **Runnable**
  → a chamada a start() aloca memória para a thread e chama run() (num novo fluxo de execução). Neste estado a thread pode ser escolhida pela JVM para executar no CPU. Java não tem estado Running.
- **Blocked**
  → thread espera por adquirir um lock
- **Waiting**
  → thread espera por ação de outra thread (ex.: join())
- **timed Waiting**
  → idêntico a Waiting mas com um tempo máximo de espera.

Decorar: estados threads
new
runnable
blocked
waiting
timed waiting



fork() e exec()

- fork() duplica todas as threads ou apenas aquela em que foi executado
  - → comportamento normal é que, após o fork(), o processo filho só tem uma thread.
  - → usar com cuidado pois podem surgir vários problemas
    - ⇒ memória inconsistente, semáforos bloqueados, etc.
    - ⇒ depois de fork() apenas funções async-safe devem ser usadas (ex.: não usar malloc() ou printf())
  - → int pthread_atfork (void(*prepare)(void), void(*parent)(void), void(*child)(void));
  - → alguns sistemas UNIX têm 2 versões que permitem escolher o comportamento (fork() e forkall()). interessante.

- em geral, exec() substitui todo o processo incluindo todas as threads. → !!!

Cancelamento de threads

- cancelar uma thread antes desta terminar por si

- 2 abordagens
  - → cancelamento assíncrono
    - ⇒ thread é terminada imediatamente
  - → cancelamento síncrono
    - ⇒ thread verifica periodicamente se deve terminar

Atendimento de sinais

- Sinais são usados em UNIX para notificar processos de certos eventos

- opções:
  - → sinal é enviado apenas para a thread a que o sinal se aplica (ex.: divisão por zero, etc.)
  - → sinal enviado para todas as threads
  - → sinal enviado para subconjunto das threads
  - → thread específica recebe todos os sinais
    - ⇒ pthread_sigmask() permite definir quais os sinais que cada thread pode receber. Assim a aplicação pode bloquear os sinais para todas as threads exceto uma (que fica com essa responsabilidade).

Thread Pools

- gerir um conjunto de threads previamente criadas, atribuindo trabalho à medida que for necessário.

- vantagens:
  - → potencialmente + rápido do que criar threads à medida que for necessário
  - → limita/controla o # de threads do sistema.

- em Java podem ser geridas através de Executor interface

Executor Interface

- permite um nível de abstração superior ao criar e manter várias threads em execução.

- se r é um Runnable, então o código

```
(new Thread(r)).start()
```

pode ser substituído, usando o executor, por:

```
e.execute(r),
```

- a execução do método run() de r pode não ser imediata e depende da política associada ao executor.

Static Executors

- O Java contém algumas implementações de Thread Pools prontas a ser usadas.

- Estas Thread Pools podem ser criadas através de métodos static de java.util.concurrent.Executors

- alguns exemplos:
  - → newSingleThreadExecutor()
    - ⇒ executa uma tarefa de cada vez

→ newFixedThreadPool(int nThreads)
⇒ thread Pool com um # fixo de threads

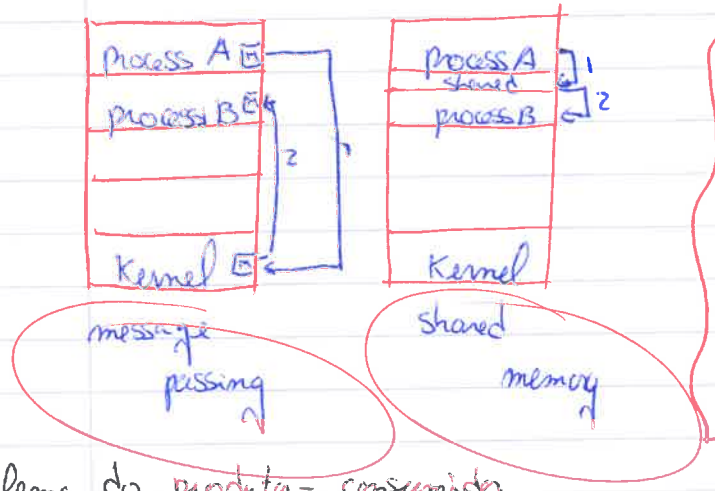
tab →|
→ newCachedThreadPool()
⇒ threads sobrevivem durante algum tempo após tarefa terminar, mas são descartadas se não forem reutilizadas após esse tempo.

Windows threads

Linux Threads

Ver nos slides

Comunicação entre processos / threads



Saber:
message passing: a comunicação entre processos usando o Kernel.

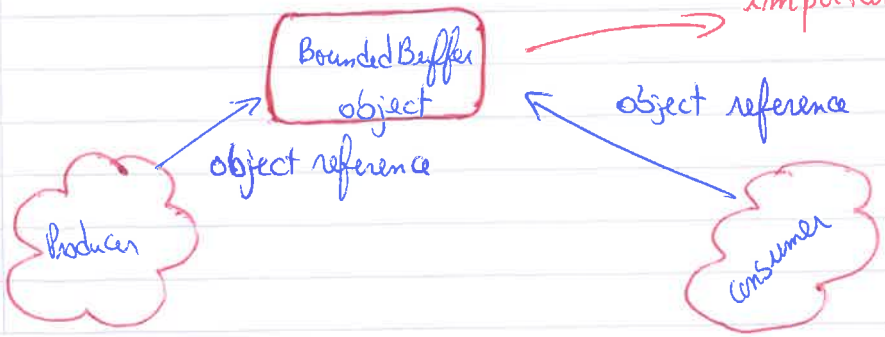
shared memory: região de memória partilhada por ambos os processos, estes trocam info escrevendo e lendo data desta região partilhada

Problema do produtor - consumidor

- paradigma para processos cooperativos
  - → processo produtor produz info
  - → info é consumida pelo processo consumidor

- um buffer partilhado armazena a info em trânsito
  - → buffer sem limites (unbounded buffer) indica que não existe limite no tamanho do buffer.
  - → buffer limitado considera que o buffer tem um tamanho fixo.

importante mas intuitivo



Ver código da solução nos slides

Definições

- condição de corrida → muito importante!
  → quando vários processos/threads acedem a ~~todos~~ dados partilhados e o resultado final depende de forma inesperada da ordem de execução.
  → processos/threads são executados pela ordem de chegada.
- região crítica
  → zona de código que manipula dados partilhados e que não pode ser executada concorrentemente por mais do que um processo/thread.

Condições para Região Crítica

- exclusão mútua
  → se um processo Pi está a executar na sua região crítica então nenhum dos outros processos pode estar em execução nas suas regiões críticas.

- progresso
  → se nenhum processo está em execução em regiões críticas e pelo menos um processo pretende o acesso à região crítica então a seleção do processo que deverá ter acesso a esta região não pode ser adiada indefinidamente.

- espera limitada
  → deve existir um limite ao # de vezes que é concedido o acesso a outros processos à região crítica, após um determinado processo ter pedido esse acesso e até que esse pedido seja satisfeito

- não há ~~###~~ nenhum pressuposto sobre a velocidade ou # de CPUs

Resumindo: Condições para Região Crítica: → exclusão mútua
→ progresso
→ espera limitada.

Definições

- região de entrada
  -> código que realiza o pedido de acesso à região crítica
- região de saída
  -> código executado após a saída da região crítica

Estrutura típica

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

Soluções por software

- variável partilhada de lock
  -> valor = 0 se Região Crítica não está a ser usada
  -> valor = 1 se Região Crítica está a ser usada

Alternância estrita

- variável turn controla acesso à região crítica

processo 0

```
while (TRUE){
    while (turn != 0)  /* loop */
    critical_region();
    turn = 1;
    noncritical_region();
}
```

processo 1

```
while (TRUE){
    while (turn != 1) /* loop */
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Algoritmo de Peterson

- 2 processos (pode ser generalizado para +)
- 2 variáveis partilhadas
  - → int turn - usada para indicar de quem é a vez de entrar (em caso de conflito)
  - → boolean flag[2] - usada para indicar que processo pretende acesso à região crítica
- 2 processos: i e j
- código para processo i:

```
While (true) {

    flag[i] = TRUE;
    turn = j
    While (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}
```

Soluções Hardware

Ver código nos slides

Semáforos

Semáforos

- ferramenta de sincronização que não necessita de busy waiting
  - ↳ espera ativa → técnica em que um processo que verifica uma condição repetidamente até que esta seja verdadeira

- Semáforo S
  - → variável inteira
  - → 2 métodos de acesso
    - ⇒ acquire() / release(); down() / up(); V() / P()

```
acquire() {
    while value <= 0
        ;
    value--;
}

release() {
    value++;
}
```

- Semáforos podem considerar que:
  - → variável inteira pode tomar qualquer valor inteiro
  - → variável interna é binária
    - ⇒ por vezes estes semáforos são designados de mutexes

```
Semaphore S = new Semaphore();
S.acquire();
    // critical section
S.release();
    // remainder section
```

Implementação de Semáforos

- quando não é possível terminar acquire imediatamente, o semáforo, em geral, bloqueia o processo numa fila de espera própria
  - → Block - bloqueia o processo que tem de esperar pelo semáforo
  - → Wakeup - acorda um / vários processos da fila de espera

- deve garantir que não existem 2 processos a executar acquire ou release simultaneamente
  - → estas funções constituem regiões críticas

```
acquire(){
    value--;
    if(value<0){
        add this process to list
        block;
    }
}
```

```
release(){
    value++;
    if(value<=0){
        remove a process P from list
        wakeup(P);
    }
}
```

Deadlock e adiamento indefinido

- Deadlock ⟶ !!!
  - → 2 ou + processos estão bloqueados à espera de um evento que apenas pode ser despoletado por um dos processos em bloqueio
  - → se S e Q forem 2 semáforos inicializados a 1

P0	P1
S.acquire()	Q.acquire()
Q.acquire()	S.acquire()
...	...
Q.release()	S.release()
S.release()	Q.release()

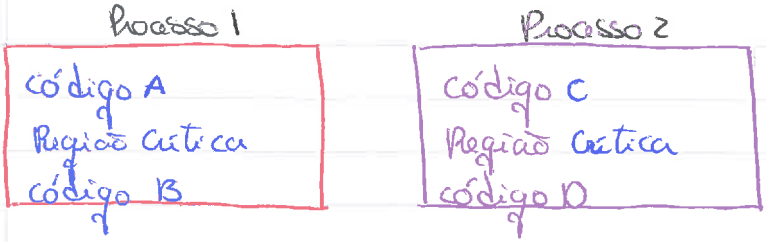
Quando P0 executa S.acquire() e P1 executa Q.acquire, depois quando P0 executa Q.acquire(), tem tem que esperar que P1 execute Q.release. Semelhantemente, quando P1 executa S.acquire() tem que esperar que P0 execute S.release().

- Adiamento indefinido (starvation) origina-se deadlock.
  - → um processo pode nunca ser removido da fila de espera de um semáforo.

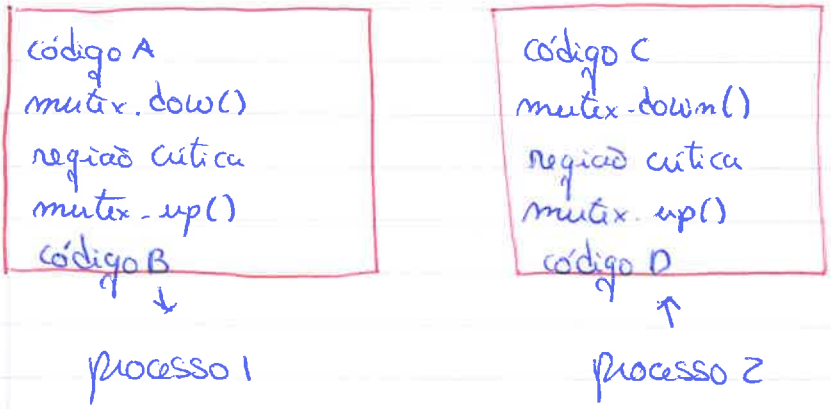
- [illegible]
- [illegible]
- [illegible]
  - → [illegible]

Exclusão mútua

- mecanismo básico de sincronização através do qual se garante o acesso em exclusão mútua a determinadas zonas de código (região crítica).



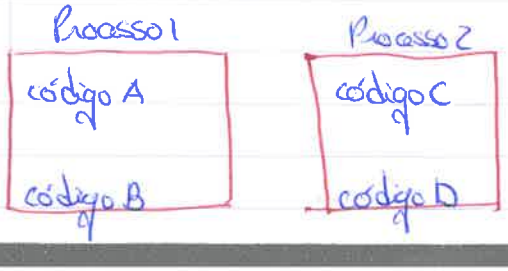
- implementação usando semáforos
  - -> usando 1 semáforo (mutex)
  - -> semáforo inicializado com valor 1
  - -> processos executam mutex.down() antes da região crítica e mutex.up() depois da mesma.



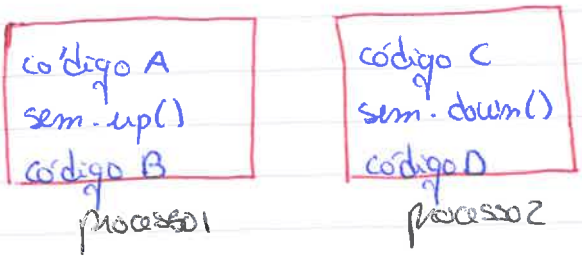
Signaling

- mecanismo básico de sincronização através do qual um processo avisa outro de que algo aconteceu.

- permite impôr que determinadas secções de código sejam precedidas pela execução de secções de código em processos distintos.
  - -> serialização de código em processos distintos
  - -> ex.: código D apenas pode executar depois de código A.



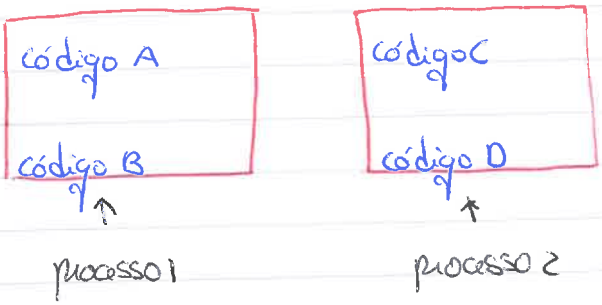
- implementação usando semáforos
  - → 1 semáforo (sem) é suficiente
    - ⇉ semáforo inicializado com valor 0
    - ⇉ processo 2 faz down() do semáforo antes de código D
      - ⇝ garantindo que espera por um up
    - ⇉ processo 1 faz up() depois de código A
      - ⇝ sinalizando processo 2 de que pode executar D.



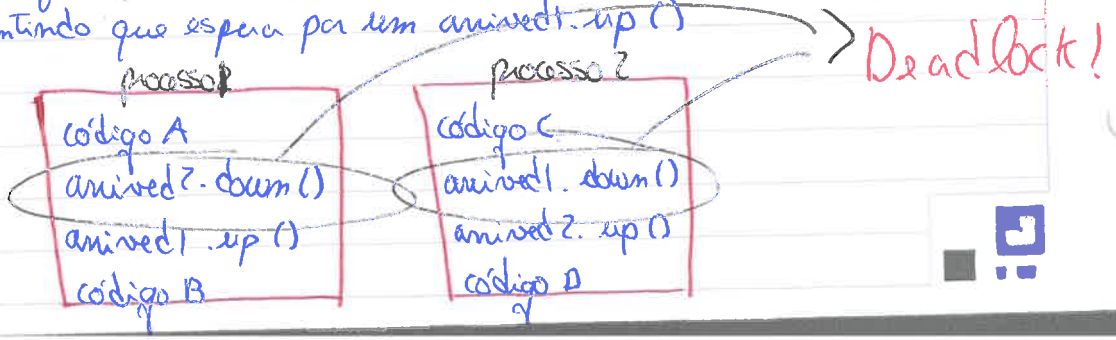
Rendezvous

- mecanismo básico de sincronização através do qual 2 processos se "encontram" antes de continuar.

- permite sincronizar determinadas operações em processos distintos
  - → ex.: processos 1 e 2 apenas avançam para código B e código D se código A e código C estiverem concluídos.

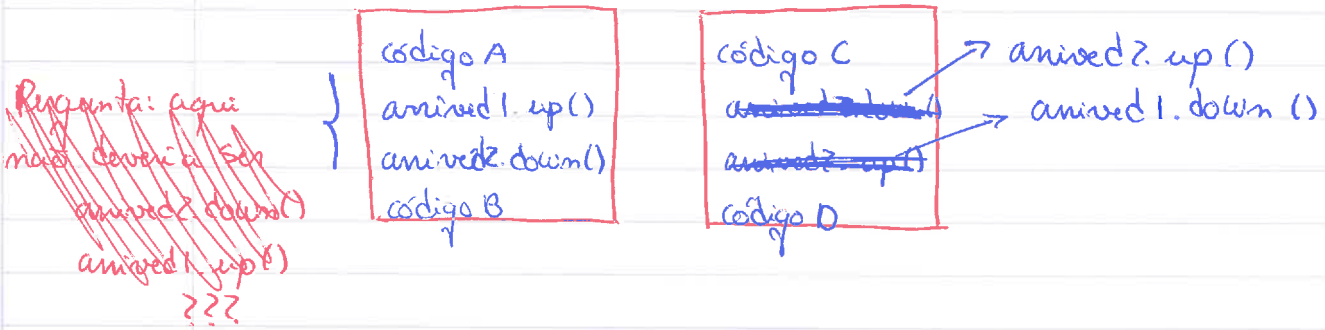


- implementação usando semáforos:
  - → usando 2 semáforos (arrived1 e arrived2)
    - ⇝ semáforos inicializados com valor 0
    - ⇝ processo 1 faz arrived1.up() e arrived2.down() após código A
      - ⇝ garantindo que espera por um arrived2.up()
    - ⇝ processo 2 faz arrived2.up() e arrived1.down() após código C
      - ⇝ garantindo que espera por um arrived1.up()



Rendezvous - maneira correta

- usando 2 semáforos (anived1 e anived2)
  - → semáforos inicializados com valor 0



Barreira

- generalização de Rendezvous para + do que 2 processos
- solução anterior de Rendezvous não é generalizável
- solução genérica
  - → 1 semáforo para exclusão mútua (mutex), 1 semáforo para barreira (barrier), 1 inteiro partilhado (count).
  - → mutex inicializado com valor 1, barrier com valor 0
  - → count inicializado com valor 0.

- Processo j, j∈1...N:

```
código j.A
mutex.down()
bool localBK = false;
if (count == N-1) {
    for(i=1..N-1) barrier.up()
    count = 0;
} else {
    count++; localBK = true;
}
mutex.up()
if (localBK) barrier.down()
código j.B
```

- funciona apenas 1 vez.
- pode dar problemas se a barreira for cíclica

porquê?

• Solução genérica + simples
→ 1 semáforo para exclusão mútua (mutex), 1 semáforo para barreira (barrier), 1 inteiro partilhado (count).

Processo j, j ∈ 1..N : código j.A
```
mutex.down()
count++;
if (count == N) {
    for (i=1..N) barrier.up()
    count=0;
}
mutex.up()
barrier.down()
código j.B
```
• funciona apenas 1 vez.
• pode dar problemas se barreira for cíclica.

• Solução genérica
→ 1 semáforo para exclusão mútua (mutex), N semáforos para barreira (array barrier), 1 inteiro partilhado (count).

Processo j, j ∈ 1..N : código j.A
```
mutex.down()
count++;
if (count == N) {
    for (i=1..N) barrier[i].up()
    count=0;
}
mutex.up()
barrier[j].down()
código j.B
```

• Solução genérica
→ 1 semáforo para exclusão mútua (mutex), 2 semáforos para barreira (array barrier), 2 inteiros partilhados (count e turn).

Processo j, j ∈ 1...N: código j. A

```
count++; local turn;
if (count == N) {
    for (i=1..N) barrier[local].up();
    count = 0; turn = 1 - turn;
}
mutex.up();
barrier[local].down();
código j. B
```

Bouded Buffer

- para implementar um Bouded Buffer com capacidade N, podem ser usados 3 semáforos:
  - → mutex: para garantir a exclusão mútua no acesso à região crítica.
  - → empty: cujo valor inteiro indica o # de espaços vazios.
  - → full: cujo valor inteiro indica o # de espaços ocupados.

Ver códigos nos slides

Escritores e leitores

- mecanismo de sincronização através do qual:
  - → existem dados partilhados
  - → podem ocorrer várias leituras em simultâneo, desde que não estejam a ocorrer escritas.
  - → durante a escrita não podem existir leituras, nem outras escritas concorrentes.

processos leitores	processos Escritores
readStart()	writeStart()
read access	write access
readEnd()	writeEnd()

- implementação usando semáforos:
  - → usando 2 semáforos (mutex e nobody) e um inteiro (readers)
  - → mutex e nobody inicializados a 1
  - → readers inicializado a 0

processos leitores

~~processos leitores~~

```
readStart()
read access
readEnd()
```

processos Escritores

```
writeStart()
write access
writeEnd()
```

- usando 2 semáforos (mutex e nobody) e um inteiro (readers)

processos leitores

```
readStart()
    mutex.down()
    if(readers == 0)
        nobody.down()
    readers++
    mutex.up()
read access
readEnd()
    mutex.down()
    readers--
    if(readers == 0)
        nobody.up()
    mutex.up()
```

processos Escritores

```
writeStart()
    nobody.down()
write access
writeEnd()
    nobody.up()
```

- Deadlock impossível
- adiamento indefinido de escritas possível

Jantar de filósofos

- problema clássico de sincronização
  - → mesa redonda, 5 filósofos, 5 garfos
  - → filósofos alternam entre pensar e comer
  - → apenas conseguem comer se tiverem 2 garfos

- processo filósofo:

```
While (true)
    think()
    getForks()
    eat()
    putForks()
```

- getForks() e putForks() devem respeitar:
  - → apenas 1 filósofo pode segurar 1 dado garfo
  - → deadlock deve ser impossível
  - → nenhum filósofo deve morrer à fome (adiamento indefinido)
  - → deve ser possível que + do que 1 filósofo coma ao mesmo tempo

- usando 5 semáforos (array forks)

→ processo filósofo f:

```
While (true)
    think()
    getForks()
        forks[ left(f) ].down()
        forks[ right(f) ].down()
    eat()
    putForks()
        forks[ left(f) ].up()
        forks[ right(f) ].up()
```

deadlock possível

Deadlock

- quando ocorre deadlock há 4 condições que se verificam:
  - → condição de exclusão mútua → cada recurso ou está livre ou foi atribuído a 1 e só 1 processo.
  - → condição de espera com retenção → cada processo, ao requerer um novo recurso, mantém na sua posse os recursos anteriormente solicitados.
  - → condição da não libertação → ninguém, a não ser o próprio processo, pode decidir da libertação de um recurso que lhe tenha sido atribuído.
  - → condição de espera circular → forma-se uma cadeia circular de processos e recursos em que cada processo requer um recurso que está na posse do processo seguinte na cadeia.

Jantar de filósofos - outra maneira de resolver

- usando 5 semáforos (array forks) e 1 semáforo (limit 4)
- limit 4 impede que os filósofos tentem pegar em garfos ao mesmo tempo.

```
→ processo filósofo f
    While(true)
        think()
        getForks()
            limit4.down()
            forks[left(f)].down()
            forks[right(f)].down()
        eat()
        putForks()
            forks[left(f)].up()
            forks[right(f)].up()
            limit4.up()
```

Monitores

- [redacted]
- abstração de alto nível usada para sincronização de processos
- apenas um processo pode estar ativo no monitor de cada vez
- é constituído por:
  - → estrutura de dados interna
  - → código de inicialização
  - → primitivas de acesso

Variáveis de condição

- permitem bloquear um processo até que determinada condição se verifique.
- 2 operações:
  - → wait() – bloqueia o processo/thread e liberta o monitor, permitindo que outro processo/thread execute primitivas do monitor.
  - → signal() – acorda um dos processos (se existir) bloqueado nesta variável de condição; se não existir processo bloqueado nada acontece.

Resolução de signal

- modelos de resolução após a execução de signal:
  - → monitor de Hoare:
    - → thread que invoca signal é colocada fora do monitor para que a thread acordada possa prosseguir.
    - → muito geral, mas a sua implementação exige uma stack, onde são colocadas as threads postas fora do monitor por invocação de signal.

→ monitor de Brinch Hansen
  → thread que invoca signal liberta imediatamente o monitor (signal é a última instrução executada);
  → simples de implementar, mas pode tornar-se bastante restritivo porque permite apenas a execução de um signal em cada invocação de uma primitiva de acesso.

→ monitor de Lampson / Redell
  → thread que invoca signal prossegue a sua execução, a thread acordada mantém-se fora do monitor e compete pelo acesso a ele
  → simples de implementa, mas pode originar situações em que algumas threads são colocadas em adiamento indefinido.

Programando com monitores

- identificar objetos partilhados
  → definir a sua interface
  → identificar estado interno e invariantes
  → implementar métodos de manipulação.

- passos para cada objeto partilhado
  → criar um lock
  → adicionar código para adquirir e libertar lock.
  → identificar e adicionar variáveis de condição.
  → adicionar loops nos waits das variáveis de condição
  → adicionar signal e broadcast.

- estrutura insistente
- usar apenas locks e variáveis de condição para a sincronização
- adquirir lock sempre no início do método e libertar sempre no fim
- ter sempre o lock quando se opera sobre variáveis de condição.
- esperar sempre num ciclo while quando o wait é invocado
- não usar sleep() para esperar por outras threads

Sincronização em Java

- primitivas de sincronização estão incluídas na própria linguagem Java
- cada objeto Java tem associado um lock
- o lock é adquirido ao entrar num método synchronized
- o lock é libertado ao sair desse método
- threads que têm de esperar são colocadas no entry set

- cada objeto tem um wait set
- quando uma thread entra num método synchronized e verifica que não pode prosseguir então pode executar wait()
  - → thread liberta o lock do objeto
  - → é bloqueada
  - → é colocada no wait set do objeto

- uma outra thread pode invocar notify() (ou notifyAll()) para retirar threads do wait set
  - → uma thread T é retirada do wait set e colocada no entry set
  - → T é colocado no estado Ready

Escalonador do CPU

- seleciona de entre os processos Ready qual o que irá ser executado no(s) CPU(s)
- escalonador é activado quando o processo:
  - → ① muda do estado de running para waiting
  - → ② muda do estado running para ready
  - → ③ muda do estado waiting para ready
  - → ④ termina
- os escalonadores que usam apenas ① e ④ são designados non preemptive
- escalonadores que usam ② e ③ são preemptive

- dispatcher encarrega-se de colocar o processo selecionado pelo escalonador em execução do CPU.
  - → mudança de contexto
  - → alterar CPU para modo de utilizador
  - → saltar para instrução do programa que permite continuar a execução do processo selecionado

- dispatch latency – tempo que o Dispatcher demora entre parar um processo e reiniciar o processo selecionado pelo escalonador.

Avaliação do escalonamento

- utilização do CPU
  → manter CPU ocupado

- débito
  → # de processos que terminem por unid. de tempo

- tempo do processo (turnaround time)
  → tempo entre submissão do processo até este terminar

- tempo de espera
  → tempo que o processo está à espera no estado Ready.

- tempo de resposta
  → tempo entre pedido e 1ª resposta (eventualmente parcial) a esse pedido

Escalonamento FCFS → non-preemptive

- First-come, First-served

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Se os processos chegarem pela ordem 1,2,3, então:

- tempo de espera: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

- tempo médio de espera: $(0 + 24 + 27)/3 = 17$

- mas se os processo chegarem pela ordem 2,3,1, então:

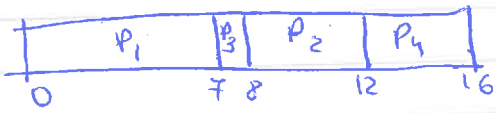
- tempo de espera: $P1 = 6$; $P2 = 0$; $P3 = 3$
- tempo médio de espera: $(6+0+3)/3 = 3$

Escalonamento SJF

- shortest Job First
- ordena os processos considerando a duração do próximo CPU burst. Executa 1º os processos com CPU burst + curtos.
- 2 opções:
  - → Nonpreemptive - uma vez atribuído o CPU ao processo fica em Running até terminar o CPU burst.
  - → Preemptive - se um processo entra na fila de Ready com um CPU Burst menor do que o tempo restante do CPU burst do processo em execução, atribuir o CPU ao processo que entrou em Ready. Também conhecido como Shortest-Remaining-Time-First (SRTF)
- SJF é ótimo do ponto de vista do tempo médio de espera de um conjunto de processos.
- SJF não pode ser implementado ao nível de short term CPU scheduling.

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (~~non~~ non-preemptive)



- tempo médio de espera = (0 + 6 + 3 + 7)/4 = 4

- SJF (preemptive)

igual ?

- tempo médio de espera = (9 + 1 + 0 + 2)/4 = 3*

* explicação: P_1 ~~chega~~ chega a 0, corre e depois espera 9 (11-2), P_2 chega a 2, espera 0 e depois espera 1 (5-4), P_3 chega a 4 e espera 0, P_4 chega a 5, mas começa a 7, espera 2 e depois não espera +.

Determinar o tempo do próximo CPU Burst

- os tempos dos CPU Burst não são, em geral, conhecidos
- solução: tentar obter boas estimativas.
- como?
  - → usar histórico do processo para prever o futuro
  - ~~exemplo~~

Escalonamento por prioridades

- priority scheduling
- é associado um nível de prioridade (int) com cada processo
  - → não existe acordo sobre se a prioridade + alta corresponde a valores baixos ou altos do nível de prioridade.
  - → iremos assumir que #s baixos representam maior prioridade.
- o CPU é atribuído ao processo com maior prioridade:
  - → preemptive
  - → nonpreemptive.

- SJF é um caso particular de escalonamento por prioridades
- problema: adiamento indefinido
  → processos com prioridade baixa podem nunca executar

- solução: contar com o tempo de espera (aging)
  → aumentar a prioridade dos processos em espera à medida que o tempo passa.

Round Robin

- versão Time sharing e preemptive de ~~FCFS~~ FCFS

- cada processo pode usar o CPU, no máx., por determinado tempo (time quantum). Se o processo não bloquear antes do tempo definido é retirado de execução e passa para o fim da lista de Ready.
  → time quantum varia, em geral, entre 10 e 100 ms.

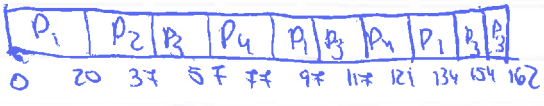
- Se existem n processos na fila de Ready (nenhum em execução) e o time quantum é q então:
  → cada processo usa cerca de 1/n do processador.
  → um processo nunca espera mais do que (n-1) q unids de tempo

- desempenho
  → Q grande ⇒ FCFS
  → q pequeno ⇒ o overhead da mudança de contexto pode ser significativo

Round Robin com q = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- O escalonamento será:



FIFO multi-nível

- multilevel queue

- fila de Ready é dividida em 2:
  - → Foreground (interativa)
  - → Background (batch)

- Fila cada tem / pode ter a sua política de escalonamento
  - → Foreground - RR
  - → Background - FCFS

- Escalonamento entre as 2 filas
  - → baseado em prioridades fixas
  - → só executa processos em background se fila Ready do foreground estiver vazia.

- divisão do tempo (Time slice)
  - → cada fila tem um certo tempo de CPU disponível
    (ex.: 80% RR e 20% FCFS)

FIFO multi-nível com realimentação

- Multilevel Feedback Queue

- um processo pode mover-se entre as várias filas

- parâmetros
  - → # de filas
  - → algoritmo de escalonamento de cada fila
  - → algoritmos de elevar e baixar a prioridade de um processo
  - → algoritmo de atribuição de prioridade inicial de um processo

- 3 filas:
  - → Q0 - RR com time quantum 8 ms
  - → Q1 - RR com time quantum 16 ms
  - → Q2 - FCFS

- Escalonamento
  - → um novo processo começa em Q0. Se esgota os 8 ms

antes de bloquear, passa para Q1
-> em Q1, se o processo ao executar esgota 16 ms antes de bloquear passa para Q2.

Linux scheduler

- o Linux considera 3 classes "clássicas" de scheduling, ordenadas por ordem decrescente de prioridade:
  - -> SCHED_FIFO - classe formada por processos cuja atribuição do processador só lhes é retirada quando processos da mesma classe, com prioridade mais alta, estão prontos a serem executados (priority superseded)
  - -> SCHED_RR - classe formada por processos cuja atribuição do processador está condicionada a uma janela de execução, a atribuição do processador é-lhes retirada + cedo quando processos da classe SCHED_FIFO, ou da mesma classe com prioridade + alta, estão prontos a serem executado (priority superseded).
  - -> SCHED_OTHER - classe formada pelos processos restantes, o processador só é atribuído a processos desta classe se não houver outro tipo de processos prontos a serem executados.
- as classes SCHED_FIFO e SCHE_RR estão associadas a processamento de tempo real e a processos de sistema e o valor das suas prioridades é fixo.
- a classe SCHED_OTHER está associada aos processos utilizador.

- foram recentemente incorporadas no linux novas classes de scheduling.
  -> SCHED_DEADLINE - classe formada por threads de tempo real, para cada thread são indicados: período, deadline relativa a tempo de computação;
  escalonada usa algoritmo Global Earliest Deadline First,

  -> SCHED_BATCH - escalonada assume que processos nesta classe são cpu-bound, usa timeslices maiores; aplica penalty quando processo acorda.

  -> SCHED_IDLE - classe formada por processos de muito baixa prioridade; o valor nice não tem efeito neste processos

  através do comando nice um utilizador pode baixar a prioridade de um processo

Earliest Deadline First

- escolhe para execução sempre o processo que tem a deadline + próxima.

- é ótimo do ponto de vista de que se é possível correr um conjunto de processos (com tempo de chegada, tempo de processamento e deadline) de forma a todos cumprirem as deadlines, o EDF cumpre as deadlines.

Memória Virtual

- ~~eficiência da~~
- eficiência da utilização da memória
  -> memória deve ser partilhada pelos processos
  -> manter em memória apenas o necessário
  -> endereços usados pelos processos não são endereços da memória física.

- Segurança
  → mecanismos de segurança que impeçam que um processo altere as

- Transparência
  → processo tem acesso a muita memória (e eventualmente + do que a memória física)
  → processo cree como se toda a memória lhe pertencesse

- partilha de memória
  → vários processos acedem à mesma zona de memória (de forma controlada).

- cada processo cree num espaço de endereçamento virtual (igual para todos).

- os endereços que o processo usa são virtuais, o mesmo endereçamento virtual de 2 processos pode corresponder a endereços físicos distintos

- os endereços da memória virtual têm de ser convertidos em endereços de memória física.

- alguns endereços de memória virtual podem estar armazenados em disco.

Mapeamento Virtual - Físico

- é necessária a conversão (rápida) do endereço virtual para o endereço.

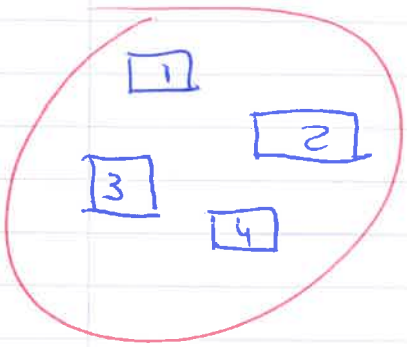
Paginação e segmentação

- **paginação**
  - → memória dividida em páginas
  - → páginas têm tamanho fixo
  - → fragmentação interna
  - → um processo necessita de um certo # de páginas livres

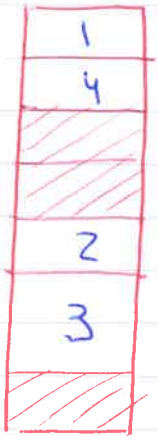
- **segmentação**
  - → memória dividida em segmentos
  - → segmentos podem ter tamanho variável
  - → fragmentação externa
  - → um processo necessita de uma zona contígua de memória livre.

paginação → memória dividida em páginas
segmentação → memória dividida em segmentos

Segmentação



user space



physical memory space

Características de memória partilhada

- O custo de um page fault é muito elevado
- páginas são relativamente grandes para amortizar o tempo de acesso.
  - → 4 KB a 64 KB.
- usar uma organização completamente associativa
- page faults são tratadas por software pois o atraso principal é o acesso ao disco.
  - → algoritmos + sofisticados

- Write-back

Encontrar uma página

- organização completamente associativa
  - → pág. virtual pode ser mapeada em qualquer pág. física.
  - → reduzir page faults.

- não é possível usar procura associativa

- tabela relaciona pág. virtual com a sua posição.
  - → page table
  - → indexada por nº pág. virtual
  - → entrada indica posição real da pág. virtual
  - → cada processo tem a sua tabela de pág.

Page fault

- valid bit da tabela de pág. com 0

- tratado pelo sistema operativo

- estrutura mantém posição das páginas virtuais em disco.

- política de substituição tipo LRU aproximado
  - → tabela de páginas contém reference bit
  - → periodicamente reference bits colocados a zero
  - → se página é acedida (touched) reference bit a 1
  - → substituir páginas com reference bit a 0.

Tamanho da tabela de página

- considere um sistema com endereços de 32 bits, páginas de 4 KB e 4 bytes por entrada na tabela de pág. Qual o espaço ocupado pela tabela de pág.?

  # de entradas = $2^{32} / 2^{12} = 2^{20}$

  tamanho da tabela = $2^{20} * 4 = 4$ MB

100 processos implicaria 400 MB de memória usados em tabelas de pág.!

Tamanho da tabela de página

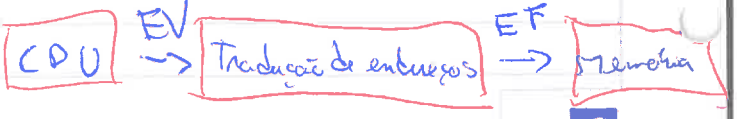
- técnicas para reduzir tabela de pág.
  - → registo limita o tamanho da tabela
    - ⇝ memória associada ao processo cresce num só sentido
  - → 2 tabelas de pág. por processo com 2 registos limite
    - ⇝ 2 segmentos (stack e heap) crescem em direções opostas
  - → hashing
    - ⇝ tabela de pág. depende do tamanho da memória física
  - → tabelas de pág. com vários níveis
  - → tabelas de pág. em memória virtual

Política de escrita

- Write-through não é eficiente
- Write-back
  - → escrita em disco pág a pág.
    - → pág. só é escrita em disco quando necessita de ser retirada da memória física...
- Dirty bit
  - → e só se foi alterada desde que foi lida no disco
- Write allocate
  - → escrita numa pág. que não reside em memória física, carrega essa pág. para a memória física e escreve.

Problemas

- memória virtual obriga a 2 referências à memória para aceder a um endereço virtual
  - → tabela de pág. reside em memória.
  - → acesso à tabela de pág.
    - → acesso à memória física



Soluções

- os acessos à memória virtual contêm localidade espacial e temporal
- cache de endereços traduzidos recentemente
- TLB - Translation-lookaside buffer
- Entrada do TLB pode incluir tag, pág. física e reference, dirty, access bits
- em cada acesso o TLB é verificado
- em cada acesso o TLB é verificado
  - → Hit - ~~conta~~ continua
  - → Miss - verificar a tabela de pág.
    - ⇝ podem originar (ou não) page defaults
    - ⇝ podem ser tratados por hardware ou software