TÉCNICO LISBOA

Programação de Sistemas Computacionais

Aulas de Laboratório

Licenciatura de Engenharia Electrónica (LEE) – TagusPark

Instituto Superior Técnico

Carlos Almeida

Setembro de 2014

1 Introdução

Este documento pretende ser um contributo para o acompanhamento das *Aulas de Laboratório* da disciplina de *Programação de Sistemas Computacionais* da Licenciatura de Engenharia Electrónica (LEE) a funcionar no Pólo TagusPark do Instituto Superior Técnico. Trata-se de um documento de trabalho susceptível de ser sujeito a refinamentos e evoluções que forem sendo julgadas convenientes. Essencialmente, tem como objectivo agregar num mesmo documento os diversos materiais que têm sido produzidos no contexto desta disciplina (e.g. exemplos de programas, enunciados de problemas e projectos, pequenas notas explicativas) bem como algum texto explicativo que, por vezes, era apenas transmitido de forma oral nas aulas.

Esta disciplina tem como principal objectivo procurar fazer com que os alunos se familiarizem com os conceitos fundamentais dos sistemas operativos, bem como interajam com um sistema operativo, tanto do ponto de vista da sua utilização directa (interface utilizador), como desenvolvendo programas que interajam e tirem partido das facilidades oferecidas por esse sistema (interface de programação). Em particular, para além do conhecimento da arquitectura do sistema operativo e dos seus di-

versos componentes, deverão ser capaz de desenvolver programas que manipulem os vários objectos computacionais suportados pelo sistema operativo, nomeadamente construindo programas concorrentes onde existem vários "processos" que, para além de eventualmente acederem a dispositivos, interagem entre si, efectuando operações de comunicação e sincronização.

Assim sendo, no contexto das aulas de laboratório, é normalmente feita: uma pequena revisão ou explicação mais pormenorizada de alguns conceitos introduzidos nas aulas teóricas, que contudo não dispensa o estudo das componentes bibliográficas a elas associadas; complementada com a apresentação de alguns exemplos de programas ilustrativos de diversas operações; sendo depois pedido aos alunos que adaptem e extendam esses exemplos de forma a exercitarem-se e a melhor dominarem os conceitos subjacentes. Partes destes exemplos poderão depois ser utilizados como base na elaboração do projecto final.

No que diz respeito à avaliação desta componente prática, para além da avaliação do projecto final, que inclui um exame oral, existe uma avaliação contínua, com a existência de metas intercalares, cujo objectivo é procurar fazer com que o trabalho vá sendo desenvolvido de forma gradual. Para além disso, também se procura que os vários exercícios e exemplos, bem como o enunciado do projecto final, envolvam diversos mecanismos distintos, de forma a permitir uma melhor cobertura da matéria global.

Este documento está organizado de forma a apresentar as várias matérias numa sequência semelhante à que é apresentada nas aulas de laboratório. Assim, começa-se por fazer uma breve descrição do ambiente de trabalho, com uma revisão de alguns comandos básicos, e referências às ferramentas de desenvolvimento a utilizar. Passa-se depois à descrição dos aspectos mais relevantes de vários objectos computacionais do sistema operativo (Linux), apoiada na apresentação de diversos exemplos ilustrativos da sua utilização. São depois tecidas algumas considerações gerais relativas ao projecto final (um exemplo de enunciado é apresentado em anexo), clarificando a sua estrutura e os objectivos fundamentais. Finalmente, são referidos alguns aspectos susceptíveis de virem a ser abordados no futuro.

2 Ambiente de Trabalho e Desenvolvimento

Embora o objectivo da disciplina seja fundamentalmente dominar os conceitos mais importantes relativos aos vários sistemas operativos, considerados de uma forma mais ou menos genérica, do ponto de vista prático é necessário utilizar um sistema operativo concreto. No contexto desta disciplina, a escolha recaíu sobre o sistema operativo Linux, por se tratar de um sistema Unix aberto, de grande divulgação, de distribuição gratuita, e para o qual existe um conjunto vasto de ferramentas de desenvolvimento da GNU, também de distribuição gratuita.

2.1 Sistema operativo

Como se disse acima, o sistema operativo utilizado nas aulas de laboratório é o Linux. Embora existam várias distribuições disponíveis, não é obrigatória a utilização de uma determinada distribuição específica. Os trabalhos a realizar são (pelo menos em princípio) compatíveis com as várias distribuições mais comuns. Atendendo ao encadeamento da disciplina no curso, é assumido que os alunos já possuem alguma experiência, do ponto de vista de utilizador (utilização no contexto de outras disciplinas), na utilização deste sistema operativo. Por esse motivo, não é feita uma descrição exaustiva dos comandos de sistema disponíveis, mas apenas uma breve revisão de alguns comandos fundamentais, e indicações de como obter mais informação caso seja necessário. (Obviamente que, caso este pressuposto não seja verdadeiro para alguns alunos, isso poderá implicar um maior auto-estudo por parte desses alunos, com um eventual apoio específico.)

O Linux é um sistema operativo multi-utilizador em que, como na maior parte dos sistemas operativos, cada utilizador é identificado por um nome (*username*) e uma senha (*password*). A "entrada" no sistema (*login*) implica a realização dessa identificação com sucesso. A partir desse momento, o utilizador poderá interagir com o sistema. Essa interacção poderá ser feita, quer através de uma interface gráfica (normalmente dependente da distribuição concreta), quer através de um interpretador de comandos (*shell*) associado a um terminal ("linha de comandos").

Embora os aspectos gerais de interacção com o sistema (edição, cópia e salvaguarda de ficheiros, por exemplo) possam ser feitos utilizando a interface gráfica que possa eventualmente existir, a execução dos programas a desenvolver será feita a partir da linha de comandos de um terminal. A própria interface utilizador dos programas a desenvolver deverá manter-se simples, pois esse não é um objectivo principal da disciplina.

Num sistema operativo unix (como é o caso do Linux), o sistema de ficheiros tem um papel muito importante na maior parte das interacções com o sistema operativo, quer feitas directamente pelo utilizador, quer feitas por aplicações. Como a maior parte dos sistemas de ficheiros, tem uma estrutura hierárquica, estando organizado na forma de uma árvore invertida, começando pela raiz (representada pelo caracter "`/`"), e podendo possuir vários directórios ("nós") e ficheiros regulares ("folhas"). Cada directório, que funciona como uma "pasta", pode conter um número indeterminado tanto de outros directórios, como de ficheiros regulares. Para além dos directórios e dos ficheiros regulares, existem ainda os ficheiros especiais que representam os dispositivos (permitindo o seu acesso), e que estão normalmente localizados no directório "`/dev`" (ou respectivos sub-directórios).

Um ficheiro é identificado perante os utilizadores e pode ser acedido com base num nome (que indica o caminho de acesso – *pathname*). Esse nome (caminho de acesso) pode ser absoluto ou relativo. É absoluto se começar a partir da raiz ("`/`"), e é relativo caso contrário (considerado a partir do directório corrente). Quando no caminho de acesso existem vários nomes de directórios, esses nomes individuais são separados por "barras" ("`/`"). O directório corrente pode ser representado por "`.`" e o directório acima pode ser representado por "`..`". A cada ficheiro corresponde um des-

critor de ficheiro, que no caso do sistema de ficheiros do Unix é designado por *inode*. É o descritor de ficheiro que contém informação relevante relativamente ao ficheiro, nomeadamente: a localização dos blocos de dados; as datas de criação, modificação e acesso; tamanho; identificação do dono; informação de protecção; etc. Uma descrição mais promenorizada é feita nas aulas teóricas.

Os directórios funcionam como um catálogo que faz corresponder o nome de um ficheiro ao seu *inode*. Quando se abre um ficheiro, utilizando, por exemplo, a chamada sistema `open`, o nome que indica o caminho de acesso do ficheiro é pesquisado sequencialmente na árvore de directórios fazendo corresponder cada nome individual ao seu respectivo *inode*, para depois se poder aceder aos respectivos dados. Isso é feito até se atingir o ficheiro final, ou não haver autorização de acesso devido aos mecanismos de protecção.

A forma mais comum de protecção num sistema de ficheiros Unix é feita com base na indicação dos tipos de acesso autorizados: leitura, escrita, execução (`rwx`). Esta especificação de acesso é feita para três entidades: utilizador (dono), grupo, outros. O super-utilizador (`root`) tem sempre acesso.

Alguns comandos da "shell"

Alguns comandos básicos	
comando	descrição
man	consultar páginas dos manuais
pwd	mostrar directório corrente
ls	listar directório
cd	mudar de directório
mkdir	criar directório
chmod	modificar protecções
ps	listar processos
kill	terminar processo (enviar um sinal a um processo)
cat	listar ficheiro

Um comando fundamental é o comando `man`, que permite consultar as páginas dos manuais dos vários comandos disponíveis, incluindo o próprio `man`. Com a opção "`-k`" (equivalente ao comando `apropos`) é possível especificar uma "palavra chave" para procurar quais os comandos em cuja página do manual se encontra essa "palavra chave".

Entradas / saídas padrão

A cada processo no Unix existem normalmente associados "canais/dispositivos" para as operações de entrada/saída e mensagens de erro (*standard input*, *standard output*, *standard error*). Por omissão são o teclado para entrada, e o ecrã para as saídas e mensagens de erro.

É possível, na linha de comando, fazer o redireccionamento destas entradas/saídas padrão especificando ficheiros para as substituir. Por exemplo, nos comandos que se seguem teremos:

```
$ prog < fich-entrada
```

o programa `prog` irá ler o ficheiro `fich-entrada` como se fosse a *entrada padrão* (*standard input*);

```
$ prog > fich-saida
```

o programa `prog` irá escrever no ficheiro `fich-saida` como se fosse a *saída padrão* (*standard output*);

```
$ prog >> fich-saida
```

o programa `prog` irá escrever no ficheiro `fich-saida` como se fosse a *saída padrão* (*standard output*), acrescentando ao conteúdo eventualmente já existente.

É também possível encadear 2 ou mais programas (normalmente designado por filtro) de forma que a *saída padrão* de um programa seja considerada a *entrada padrão* do programa seguinte. No exemplo seguinte:

```
$ prog1 | prog2
```

as mensagens escritas pelo programa `prog1` na *saída padrão* (*standard output*) serão lidas pelo programa `prog2` como vindo da *entrada padrão* (*standard input*).

Execução de programas

A execução de programas, tal como a execução de comandos, é feita a partir do interpretador de comandos (*shell*). A *shell* (existem várias alternativas, por exemplo: `sh`, `ksh`, `csh`, `tcsh`, `bash`) cria um processo para executar o programa e fica à espera que esse processo termine. Caso se deseje, é possível libertar a "shell" executando o programa em "background" (por oposição a "foreground"). Para isso basta juntar o caracter "`&`" no fim da linha de comando. Deste modo o processo que executa o programa funcionará em paralelo com a "shell", ficando esta disponível para executar outros comandos.

Por omissão, os vários processos que forem criados num mesmo terminal (janela), partilharão esse terminal para as operações de entrada/saída. As mensagens de escrita (saída) serão feitas livremente, podendo o ecrã ficar com mensagens intercaladas provenientes dos vários processos. No entanto, as operações de leitura (entrada) só podem ser realizadas se o processo estiver em "foreground". Se o processo tentar fazer uma operação de leitura da "entrada padrão" (teclado), enquanto está em "background", ficará bloqueado até que seja transferido para "foreground".

A transição de um determinado processo entre "backgroung" e "foreground" é possível através da utilização dos comandos `bg` e `fg`.

A terminação de um processo que esteja a ser executado em "background" acontecerá quando esse processo terminar por si, ou então será necessário enviar-lhe um sinal para terminar (`kill`). Esse envio de sinal é efectuado especificando o identificador do processo (`pid`), que pode ser consultado com o comando (`ps`).

Nota: Um programa para poder ser executado necessita que os seus atributos em termos de protecção indiquem acesso de execução (`x`). Normalmente isso é feito automaticamente quando o programa é gerado com sucesso. No entanto, se por acaso, aquando de eventual transferência de ficheiros entre computadores, essa protecção não estiver correcta, é necessário corrigi-la com o comando `chmod` para que seja possível executar o programa.

2.2 Editor, compilador e bibliotecas

Os programas são desenvolvidos utilizando a linguagem de programação C. A edição dos ficheiros com o código fonte dos programas pode ser feita com qualquer editor de texto disponível, de acordo com as preferências do utilizador. Exemplos de editores que normalmente estão disponíveis nos sistemas unix, e em particular no Linux, são: `emacs`, `vi`, `gedit`. Mas poderão existir outros.

A compilação dos ficheiros fonte ("`.c`") para produzir os ficheiros com o código objecto ("`.o`") é feita utilizando o compilador C da Gnu (`gcc`). Os vários ficheiros objecto são ligados entre si e com eventuais bibliotecas para produzir o ficheiro com o código executável final.

2.3 Ferramentas auxiliares – make

O utilitário `make` é uma ferramenta muito útil no desenvolvimento de projectos de programação pois permite definir relações de dependência, e especificação de comandos a executar, de forma a manter os programas "alvo" actualizados. Funciona com base na data/hora das últimas modificações, sendo as dependências e os comandos a executar especificados num ficheiro "makefile".

Com um simples comando serão executados em cadeia todos os comandos necessários à actualização do código (e apenas esses), tendo em conta os vários ficheiros eventualmente modificados desde a última vez, e usando os parâmetros específicos que foram configurados para o projecto concreto. Exemplo:

```
$ make ``alvo''
```

ou simplesmente (caso se pretenda considerar o primeiro alvo do ficheiro makefile):

```
$ make
```

Estrutura do ficheiro "makefile":

```
"alvo": "dependências"
<tab> "comando"
```

podendo a "resolução" das "dependências" ser feita de forma recursiva (cada dependência é considerada como um alvo).

Existe um conjunto vasto de opções e regras associadas ao utilitário `make`, mas é possível a sua utilização com uma estrutura relativamente simples, podendo ainda tirar-se partido da existência de várias regras por omissão.

Exemplo de ficheiro `makefile` ("< tab >" representa o caracter `tab`):

```
prog1: prog1.o aux.o
<tab> gcc -o prog1 prog1.o aux.o

prog1.o: prog1.c prog1.h
<tab> gcc -c prog1.c

aux.o: aux.c
<tab> gcc -c aux.c
```

No ficheiro "makefile" também é possível efectuar a definição de "flags", quer globais (reconhecidas pelo ambiente de trabalho), quer locais (para serem usadas internamente no ficheiro "makefile"). Exemplo de definição de "flags" globais (relacionadas com o compilador):

```
CC = gcc
CFLAGS = -g -Wall
LDFLAGS = -lpthread
```

Exemplo de definição de "flag" local:

```
OBJS = prog1.o aux.o
```

Exemplo de novo ficheiro `makefile`, utilizando a definição de "flags", e tendo em conta regras por omissão (chamada do compilador de C para transformar os ficheiros "`.c`" em ficheiros "`.o`":

```
CC = gcc
CFLAGS = -g -Wall
OBJS = prog1.o aux.o

prog1: $(OBJS)
<tab> $(CC) -o prog1 $(OBJS)
```

2.4 Depuramento do código – debugger gdb

No desenvolvimento de programas, o depuramento do código – testes para eliminar eventuais erros de programação – reveste-se de extrema importância, e pode ser uma tarefa morosa e bastante ingrata. A existência de ferramentas de depuramento (*debuggers*) é algo extremamente útil, podendo reduzir consideravelmente o tempo necessário a esse depuramento. Daí que seja importante o seu conhecimento, mesmo que de uma forma rudimentar, para poder utilizar quando necessário. Isso permitirá o aumento da produtividade uma vez que torna possível um maior controlo do programa. Em vez de simples mensagens para o ecrã ("printfs"), que têm um alcance limitado e podem implicar mais operações de compilação, o uso do *debugger* permite um controlo mais fino, com a possibilidade de: especificar paragens em determinados pontos do programa ("breakpoints"); execução passo-a-passo; examinar conteúdo de variáveis, etc.

O *debugger* mais comum num ambiente Unix é o `gdb` da Gnu. O conhecimento de alguns dos seus comandos básicos e das suas potencialidades é, normalmente, suficiente para uma utilização elementar. Para uma utilização mais complexa, em caso de necessidade, poderão ser consultados os respectivos manuais, quer os integrados na própria aplicação como mecanismos de ajuda (`help`), quer outros manuais mais completos com documentação específica.

O `gdb` pode ser utilizado, com a sua interface básica, a partir da linha de comando. Existe, no entanto, a possibilidade de usar interfaces mais elaboradas (gráficas) construídas sobre a versão mais elementar (e.g. `ddd`).

A tabela seguinte apresenta alguns dos comandos básicos que estão disponíveis no `gdb`, e que permitem resolver a maior parte dos problemas de depuramento de um programa:

Alguns comandos básicos do `gdb`	
comando	descrição
break[point]	colocar um ponto de paragem (breakpoint) especificando o nome da função ou número da linha de código
r[un]	executar o programa
bt	mostrar o encadeamento de chamadas (backtrace)
l	listar linhas de código
s[tep]	executar a próxima instrução (entrando nas subrotinas)
n[ext]	executar a próxima instrução (saltando as subrotinas)
print	mostrar o conteúdo da váriável especificada como argumento
c[ontinue]	continuar a execução do programa

3 Processos e Concorrência

Um *processo* é a entidade activa num sistema operativo. Está associado a um *programa*, que é um conjunto de instruções que são executadas num *processador* (CPU).

Podem existir vários processos a executarem-se "simultaneamente" (de forma concorrente). Esta "simultaneidade" pode ser real ou aparente dependendo de existir ou não mais do que um processador. No caso de haver apenas um processador pode dizer-se que existe um "pseudo-paralelismo", uma vez que num dado instante o processador apenas estará a executar uma instrução de um processo. De qualquer forma, independentemente de haver um ou mais processadores, de um ponto-de-vista mais macroscópico, é como se existissem várias máquinas virtuais a executar os vários processos. Teremos, portanto, uma situação de concorrência entre os vários processos. No caso de existir apenas um processador, isso é conseguido através da multiplexagem no tempo da utilização do CPU. Existe associado a cada processo um *contexto* (informação sobre o conteúdo dos registos, e outra informação relevante dependente do sistema operativo), havendo salvaguardas e reposições aquando da troca de processos (mudança de contexto). As decisões relativas ao momento em que a troca de processos deve ser efectuada dependem do algoritmo de *escalonamento* utilizado (ver matéria teórica).

A programação de processos concorrentes exige algum cuidado de forma a garantir o seu correcto funcionamento. O acesso a eventuais recursos partilhados tem de ser controlado de forma a garantir que esse acesso é feito de forma coerente. Por exemplo, o acesso a um dispositivo como uma impressora, ou a actualização de uma estrutura de dados partilhada, podem implicar a necessidade de um acesso em exclusividade. Isso implica a existência de *secções críticas* no código, em que é necessário garantir a exclusão mútua, ou seja, se um processo está a aceder ao recurso, nenhum outro processo poderá aceder enquanto este não terminar esse acesso.

Para além da "competição" por recursos partilhados, poderão existir processos que necessitem de "cooperar" entre si, para atingir um determinado objectivo comum. Para facilitar o desenvolvimento de aplicações concorrentes com este tipo de requisitos, é importante a existência de mecanismos de comunicação e sincronização entre processos que permitam lidar com essas situações de forma correcta. Nas secções seguintes serão referidos alguns exemplos desses mecanismos de comunicação e sincronização.

3.1 Processos e Fios-de-execução (threads)

Associado a cada processo existe um *espaço de endereçamento* que corresponde ao conjunto de posições de memória a que o processo pode aceder. Na maior parte dos sistemas operativos, esse espaço de endereçamento está protegido do acesso dos outros processos com base na unidade de gestão de memória. Por exemplo, no caso de um sistema de memória virtual paginada (ver aulas teóricas sobre gestão de memória), existe associado a cada processo uma tabela de páginas que contém os descritores que permitem aceder às várias posições de memória autorizadas a esse processo. Como cada processo tem a sua tabela de páginas, só pode aceder às posições de memória para as quais possui um descritor válido.

O espaço de endereçamento de um processo está ainda, normalmente, dividido em três regiões de memória, correspondentes ao código, dados e pilha (stack). O código poderá, eventualmente, ser partilhado com outros processos, mas as regiões

de memória correspondentes aos dados e à pilha são específicas de cada processo.

Para além do processo, existem sistemas operativos que, relativamente à concorrência, suportam também um outro tipo de entidade, que é a *thread* ou *fio-de-execução*. As *threads* permitem a existência de concorrência dentro de um mesmo espaço de endereçamento. Possuem uma *pilha* (*stack*) própria, mas partilham a mesma zona de dados. Desta forma todas as váriáveis globais podem ser acedidas directamente pelas várias *threads* existentes nesse processo. Isto permite uma maior eficiência na mudança de contexto entre threads, uma vez que a tabela de páginas é a mesma. No entanto, se essas variáveis forem actualizadas pelas várias threads, isso implica algum cuidado no seu acesso, eventualmente recorrendo à utilização de mecanismos de sincronização para garantir a coerência.

Em termos gerais, os *processos* oferecem uma maior protecção, garantindo um melhor confinamento dos acessos, evitando que um eventual erro de programação associado ao código de um processo possa por em causa a integridade dos outros processos. Por outro lado, implica que as mudanças de contexto sejam menos eficientes, uma vez que é necessário mudar as tabelas de páginas quando há mudanças de contexto. No caso das threads, como não é necessário mudar as tabelas de página, as mudanças de contexto são "menos pesadas", o que torna a operação mais eficiente. Por outro lado deixa de existir a protecção associada ao confinamento dos acessos, havendo a possibilidade de que um erro associado a uma dada thread possa afectar as outras threads do mesmo processo.

3.2 Exemplos de criação de processos

Num sistema operativo Unix, os processos são criados com a função `fork`. Em caso de sucesso, após a chamada da função `fork` passarão a existir dois processos: o processo original (processo "pai"); e um novo processo (processo "filho"). Ambos executarão o mesmo código (instrução seguinte à chamada à função `fork`), mas o novo processo terá uma nova zona de dados e uma nova zona de pilha. A zona de dados do novo processo (processo "filho") será uma cópia da zona de dados do processo original (processo "pai"). As variáveis globais terão o mesmo valor após a criação, mas passarão a ser variáveis distintas (apesar de continuarem a ter o mesmo nome), pelo que eventuais alterações por parte de um processo não afectarão o outro processo.

Os dois processos podem ser distinguidos com base no valor retornado pela função `fork`: o processo "pai" recebe como retorno o identificador (pid) do processo "filho"; e o processo "filho" recebe como retorno o valor 0 (ver ilustração seguinte):

```
pid = fork();
if (pid == 0) {
/* código executado pelo processo filho */
}
else {
/* código executado pelo processo pai */
}
```

Se se pretender que o novo processo execute uma imagem de código diferente, é necessário que ele chame uma outra função (por exemplo `execl` – existem várias variantes) em que se especifica qual é o ficheiro de código que se pretende executar, deixando então de partilhar o código com o processo "pai".

O programa seguinte corresponde a um exemplo de criação de processos usando as funções de sistema `fork` e `execl`. O processo inicial (processo "pai") cria um novo processo (processo "filho") utilizando a função `fork`. O processo "filho" irá depois executar uma imagem de código diferente (o comando `ls` – para listar o directório corrente) usando a função `execl`. Entretanto, o processo "pai" vai ficar bloqueado à espera que o processo "filho" termine (usando a função `wait`).

```
/* Exemplo de utilizacao do "fork" e do "execl"
   para criar um processo. -- CRA
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>


int main()
{
  pid_t pid;
  int status;

  printf("Ola'! Eu sou o pai. O meu pid=%d\n", getpid());

  if ((pid = fork()) == 0) {
    printf("Ola'! Eu sou o filho. O meu pid=%d\n", getpid());
    printf("O filho vai listar o directorio\n");
    execl("/bin/ls", "ls", "-l", NULL);
  }
  else {
    printf("Eu sou o pai. O filho foi criado com pid=%d\n", pid);
    printf("Pai: vou esperar\n");
    pid = wait(&status);
    printf("Pai: filho pid=%d terminou com status=%d\n", pid, status);
  }

  return 0;
}
```

3.3 Exemplos de criação de threads

A existência de suporte a *threads* pode ser assegurada quer directamente pelo próprio sistema operativo, quer através de bibliotecas que serão ligadas com a aplicação. No nosso caso é utilizada a biblioteca de threads posix – **pthreads**.

Ao contrário do que acontece na criação de processos usando `fork`, em que o novo processo começa por executar o mesmo código do processo "pai", na criação de threads

é especificado qual o código (correspondente a uma função em C) que vai corresponder a essa thread. Podem ser criadas várias threads com o mesmo código.

O programa que se segue mostra um exemplo que ilustra a criação de 3 threads (usando a função `pthread_create`) que irão executar o mesmo código (código correspondente à função `thread_func`). Apesar de executarem o mesmo código, cada thread tem a sua *pilha*, pelo que as váriáveis locais serão distintas. Também é possível passar um argumento à thread no momento da sua criação (neste caso um apontador para um identificador – `&id[i]`) que pode ser usado para passar informação específica, permitindo, por exemplo, distinguir as várias threads. O programa principal, após a criação das várias threads, fica à espera que elas terminem (usando a função `pthread_join`).

Nota: Se o programa principal terminar, o processo termina, implicando que as threads terminam também.

```
/* Exemplo de programa que cria pthreads. --CRA
 */

#include <stdio.h>
#include <pthread.h>

#define NT 3

void *thread_func(void * pi)
{
  int i = *(int *)pi;
  printf("thread_func: arg=%d\n", i);
  printf("thread_func: terminar t=%d\n", i);
  return 0;
}

int main()
{
  pthread_t  threads[NT];
  int i;
  int id[NT];

  printf("main: criar thread(s)\n");
  for (i=0; i<NT; i++) {
    id[i]=i;
    if (pthread_create(&threads[i], NULL, thread_func, (void *)&id[i]) != 0) {
      printf("Erro a criar thread=%d\n", i);
    }
  }

  printf("main: esperar fim thread(s)\n");
  for (i=0; i<NT; i++) {
    pthread_join(threads[i], NULL);
  }
  printf("main: terminar\n");

  return 0;
}
```

4 Mecanismos de Comunicação e Sincronização

Como se referiu atrás, para o correcto desenvolvimento de aplicações concorrentes, em que os vários processos competem e cooperam entre si, é necessária a existência de mecanismos de comunicação e sincronização entre processos. Estes podem ser baseados em mecanismos de troca de mensagens ou na existência de acesso a zonas de memória partilhada.

Nota: A partir deste momento, a não ser que seja necessário distinguir essa situação, iremos designar como *processo* qualquer entidade concorrente (processo ou thread). Obviamente que o acesso a variáveis partilhadas é conseguido de forma distinta, mas desde que isso não seja relevante para o caso concrecto em análise, será assumido como tendo sido tratado de forma independente.

As funções concretas para sincronização e comunicação podem depender do sistema operativo que se está a utilizar. No entanto, independentemente da sintaxe específica, existem determinados mecanismos que são mais ou menos comuns nos vários sistemas operativos. Por exemplo, do ponto-de-vista de sincronização, é comum a existência de: *semáforos, mutexes, variáveis de condição* (ver aulas teóricas).

Para tratar de uma forma mais genérica os problemas de sincronização entre processos, podemos considerar a existência das seguintes funções para efectuar operações sobre *semáforos*: `esperar` e `assinalar`. No desenvolvimento de um programa concreto, num determinado sistema operativo, serão depois substituídas pelas funções disponíveis.

4.1 Problemas de sincronização

O desenvolvimento de aplicações concorrentes, que funcionem correctamente, não é uma tarefa simples, sobretudo para quem esteja habituado a apenas desenvolver aplicações sequenciais. As potenciais interferências entre os vários processos, se não forem correctamente tratadas, podem comprometer a coerência da aplicação. Daí que seja importante adquirir experiência na resolução de problemas de sincronização entre processos.

Nas aulas teóricas são analisados alguns exemplos clássicos de interacção entre processos: "produtores / consumidores"; "leitores / escritores"; "jantar dos filósofos". Estes exemplos permitem abordar os principais aspectos a ter em consideração na resolução de problemas de sincronização entre processos. No entanto, é importante que os alunos não se limitem a consultar problemas resolvidos, mas que os tentem resolver de forma a sentirem e lidarem com as eventuais dificuldades que possam existir.

Na resolução deste tipo de problemas, é necessário analisar qual o estado global do sistema que é necessário manter, e caso isso não seja possível apenas com as próprias funções de sincronização (por exemplo através do número de unidades nos semáforos), é necessário determinar quais as variáveis partilhadas a usar e quais os seus valores iniciais. Havendo variáveis partilhadas a ser actualizadas, é necessário garantir o seu acesso em exclusividade. Para isso poderá ser usado um mutex ou um

semáforo inicializado com uma unidade. Para além disso, é necessário determinar quais as situações de sincronização distintas existentes no problema concreto associando a cada uma delas um semáforo distinto.

O conjunto dos 3 exemplos clássicos referidos acima (cuja análise é feita nas aulas teóricas) permite chamar a atenção para algumas particulariedades que facilitam a resolução deste tipo de problemas e/ou permitem a sua optimização:

- utilização do número de unidades nos semáforos como recursos, permitindo reduzir o número de variáveis globais e simplificando o algoritmo;
- utilização genérica de variáveis globais para definir de forma correcta o estado geral do sistema;
- identificação de classes de processos correspondentes a situações distintas;
- evitar "corridas críticas" garantindo a coerência do sistema aquando do desbloqueamento de um dado processo;
- utilização de semáforos privados (um único processo a executar `esperar`) para garantir um desbloqueamento sem ambiguidades.

A seguir mostram-se 2 exemplos de enunciados de problemas de sincronização entre processos, correspondentes a uma "multiplicação de matrizes", e a um "cruzamento de uma ponte". O objectivo é os alunos praticarem a resolução deste tipo de problemas, fundamentalmente numa abordagem "papel e lápis", e usando as funções genéricas `esperar` e `assinalar`. No entanto, os problemas e respectivas soluções podem depois ser adaptados para serem programados utilizando pthreads e os mecanismos de sincronização a elas associados.

Problema 1: Considere uma aplicação concorrente "MultMat"que efectua a multiplicação de 2 matrizes A e B (de dimensão MxM) colocando o resultado numa matriz C. Para isso, cria P processos "MultLinha"(P¡M) que efectuam em paralelo a multiplicação de linhas. Cada processo "MultLinha"vai tentando, continuamente, multiplicar novas linhas até estarem todas calculadas. Nessa altura sinaliza o seu fim e termina. O programa principal "MultMat"é responsável pelas inicializações e pela criação dos processos "MultLinha", ficando depois à espera que estes notifiquem a sua terminação.

Considerando que tem disponíveis a primitiva "CriarProcesso"para criar os processos "MultLinha", e as primitivas"CriarSemaforo", "Esperar"e "Assinalar"para efectuar operações sobre semáforos, e que tem acesso a variáveis partilhadas, programe, utilizando a linguagem C, o código de "MultMat"e de "MultLinha". Considere apenas os aspectos relativos à sincronização e não se preocupe com o código relativo à multiplicação de matrizes.

Problema 2: Considere que pretende realizar um sistema para controlar o acesso de várias viaturas a uma ponte. Devido às características e idade da ponte, existem algumas restrições na sua utilização. Nomeadamente, apenas se pode circular num único sentido de cada vez (Norte-Sul ou Sul-Norte), e o número máximo de viaturas na ponte não pode exceder NMAX viaturas (NMAX=3).

Do ponto de vista do sistema de controlo, considere que cada viatura é um processo que, quando pretende entar na ponte, chama as funções "EntrarNS"ou "EntrarSN", consoante se desloque no sentido Norte-Sul ou Sul-Norte, respectivamente. Caso não se verifiquem as condições necessárias para entrar na ponte, ficará bloqueado nessa função. Será desbloqueado assim que puder entrar. Quando uma

viatura (processo) termina a travessia da ponte, chama a função ”SairNS”ou ”SairSN”, dependendo do sentido em que efectuou a travessia.

a)- Considerando que tem disponíveis as primitivas ”CriarSemaforo”, ”Esperar”e ”Assinalar”para efectuar operações sobre semáforos, e que tem acesso a variáveis partilhadas, programe, utilizando a linguagem C, os procedimentos ”EntrarNS”, ”EntrarSN”, ”SairNS”e ”SairSN”. Indique também quais as inicializações assumidas.

b)- Considere agora que se uma viatura chega à ponte, e já existem viaturas à espera em sentido contrário, deverá esperar, mesmo que a travessia esteja a ser efectuada no seu sentido, e o número de viaturas na ponte seja inferior a NMAX. Quando há uma mudança de sentido, serão autorizadas a entrar, até NMAX viaturas, desde que já se encontrem à espera nesse momento. Não havendo viaturas à espera em sentido contrário, uma viatura pode avançar desde que respeite o número máximo de viaturas na ponte e o sentido único.

c)- Se a restrição na utilização da ponte apenas fosse relativa ao número máximo de viaturas, sendo possível usar os dois sentidos simultaneamente, como simplicaria o algoritmo.

4.2 Exemplos de sincronização de threads com semáforos e mutexes

Os programas seguintes correspondem a exemplos que ilustram a utilização de *mutexes* e *semáforos posix* para garantir o acesso coerente a variáveis partilhadas entre *threads*, e outras situações de sincronização.

No primeiro exemplo são criadas 2 `pthreads` que irão incrementar uma variável comum (variável `cont`, declarada como variável global para poder ser partilhada pelas threads). A variável `mux` vai corresponder ao identificador de um *mutex*, sendo também uma variável global para ser acedida pelas 2 threads. O *mutex* é inicializado (função `pthread_mutex_init`) antes da criação das threads para poder ser utilizado por estas. A actualização da variável partilhada `cont`, por parte das threads, é protegida pelas chamadas às funções `pthread_mutex_lock` e `pthread_mutex_unlock`, de forma a garantir o acesso em exclusividade e assegurar a coerência.

```
/* Exemplo de programa com pthreads e mutexes. --CRA
 */

#include <stdio.h>
#include <pthread.h>

#define NT 2

pthread_mutex_t mux;
int cont=0;

void *thread_func(void * pi)
{
  int i = *(int *)pi;
  printf("thread_func: arg=%d\n", i);
  pthread_mutex_lock(&mux);
  cont++;
  printf("thread_func: t=%d cont=%d\n", i, cont);
  pthread_mutex_unlock(&mux);
  printf("thread_func: terminar t=%d\n", i);
```

```
  return 0;
}

int main()
{
  pthread_t  threads[NT];
  int i;
  int id[NT];
  if (pthread_mutex_init(&mux, NULL) != 0) {
      printf("Erro a inicializar mutex\n");
      return -1;
  }
  printf("main: criar thread(s), cont=%d\n", cont);
  for (i=0; i<NT; i++) {
    id[i]=i;
    if (pthread_create(&threads[i], NULL, thread_func, (void *)&id[i]) != 0) {
      printf("Erro a criar thread=%d\n", i);
    }
  }

  printf("main: esperar fim thread(s)\n");
  for (i=0; i<NT; i++) {
    pthread_join(threads[i], NULL);
  }
  printf("main: terminar, cont=%d\n", cont);

  return 0;
}
```

No exemplo seguinte ilustra-se a sincronização entre o programa principal e uma thread através da utilização de semáforos. São criados 2 semáforos (`sem1` e `sem2`) ambos com 0 unidades iniciais. Existem 2 pontos de sincronização entre a thread e o programa principal com base nos semáforos:

- a thread espera no semáforo `sem1` (operação `sem_wait(&sem1)`) até que o programa principal tenha assinalado esse semáforo (operação `sem_post(&sem1)`);
- o programa principal espera no semáforo `sem2` (operação `sem_wait(&sem2)`) até que a thread tenha assinalado esse semáforo (operação `sem_post(&sem2)`).

```
/* Exemplo de programa com pthreads e semaforos. --CRA
 */

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define NT 1

sem_t sem1, sem2;

void *thread_func(void * pi)
{
```

```
  int i = *(int *)pi;
  printf("thread_func: arg=%d\n", i);
  sem_wait(&sem1);
  printf("thread_func: sem1-sem2 t=%d\n", i);
  sem_post(&sem2);
  printf("thread_func: terminar t=%d\n", i);
  return 0;
}

int main()
{
  pthread_t  threads[NT];
  int i;
  int id[NT];
  if (sem_init(&sem1, 0, 0) != 0) {
      printf("Erro a inicializar semaforo 1\n");
      return -1;
  }
  if (sem_init(&sem2, 0, 0) != 0) {
      printf("Erro a inicializar semaforo 2\n");
      return -1;
  }
  printf("main: criar thread(s)\n");
  for (i=0; i<NT; i++) {
    id[i]=i;
    if (pthread_create(&threads[i], NULL, thread_func, (void *)&id[i]) != 0) {
      printf("Erro a criar thread=%d\n", i);
    }
  }

  sem_post(&sem1);
  printf("main: sem1-sem2\n");
  sem_wait(&sem2);

  printf("main: esperar fim thread(s)\n");
  for (i=0; i<NT; i++) {
    pthread_join(threads[i], NULL);
  }
  printf("main: terminar\n");

  return 0;
}
```

Finalmente, ilustra-se uma situação correspondente à extensão do exemplo anterior para incluir funcionalidades adicionais. Utilizando a sincronização base anteriormente explicada, constrói-se uma aplicação que recebe uma cadeia de caracteres (string) da entrada padrão (stdin) e converte os caracteres para maiúsculas. A leitura e escrita dos caracteres é feita pelo programa principal, mas a conversão é feita pela thread.

```
/* Exemplo de programa com pthreads e semaforos.
 * Sincronizacao no processamento de string. --CRA
 */
```

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <ctype.h>

#define DIM 11

sem_t sem1, sem2;
char buf[DIM];

void *thread_func(void * pi)
{
  int i;

  printf("thread_func: wait\n");
  sem_wait(&sem1);
  printf("thread_func: processa\n");
  for (i=0; buf[i]; i++)
    buf[i] = toupper(buf[i]);
  sem_post(&sem2);
  printf("thread_func: terminar\n");
  return 0;
}

int main()
{
  pthread_t  thread;
  if (sem_init(&sem1, 0, 0) != 0) {
      printf("Erro a inicializar semaforo 1\n");
      return -1;
  }
  if (sem_init(&sem2, 0, 0) != 0) {
      printf("Erro a inicializar semaforo 2\n");
      return -1;
  }
  printf("main: criar thread\n");
  if (pthread_create(&thread, NULL, thread_func, NULL) != 0) {
    printf("Erro a criar thread\n");
  }

  printf("buf: ");
  fgets(buf, DIM, stdin);
  sem_post(&sem1);

  sem_wait(&sem2);
  printf("buf: %s\n", buf);

  printf("main: esperar fim thread\n");
  pthread_join(thread, NULL);
  printf("main: terminar\n");

  return 0;
}
```

4.3 Exemplos de comunicação com pipe

Para 2 processos poderem comunicar entre si, necessitam de ter acesso a um mecanismo de comunicação comum, ou seja, ambos terem acesso a um identificador que corresponda ao mesmo mecanismo de comunicação. Essa "partilha" de identificador pode ser conseguida por "herança" no caso de processos que sejam dependentes hierarquicamente (pai e filho). Como se disse anteriormente, na criação de processos no Unix, com a função `fork`, o processo "filho" fica com uma zona de dados que é a cópia da zona de dados do processo "pai" no momento da criação. Para além das eventuais variáveis globais existentes, o filho herda também uma cópia da tabela de ficheiros abertos, pelo que ficheiros abertos pelo processo "pai" antes da criação do processo "filho" também poderão ser acedidos por este.

Quando os processos são independentes não existe a possibilidade de passagem de parâmetros por "herança". Nesse caso é necessário que exista alguma forma de ter nomes globais que permitam uma referência comum. Alguns exemplos serão explicados nas secções seguintes. Relativamente aos *pipes*, existe uma variante (normalmente designada por *named pipe*) que, através da utilização da função `mkfifo`, permite criar um *pipe* ao qual está associado um nome global, podendo depois ser aberto pelos processos que pretendem comunicar entre si.

O exemplo seguinte ilustra a comunicação entre 2 processos usando um `pipe`. Associado a cada `pipe` existem 2 identificadores: um para realizar operações de leitura; e outro para realizar operações de escrita. Embora à partida qualquer um dos processos possa efectuar tanto operações de escrita como de leitura, na sua utilização normal, cada processo apenas efectuará uma operação de um determinado tipo (leitura ou escrita) permitindo uma comunicação unidireccional. Caso se pretenda comunicação bidireccional deverão ser utilizados 2 *pipes*.

No exemplo, a criação do *pipe* (chamada da função `pipe(fd)`) é efectuada antes da criação do novo processo (chamada `fork()`) para, tal como se referiu acima, o novo processo herdar os respectivos identificadores. Após a execução com sucesso de `fork`, passarão a existir 2 processos: o processo "pai" e o processo "filho". O processo "filho" irá chamar a função `write(fd[1], ...)` para enviar uma mensagem para o *pipe*. O processo "pai" chama a função `read(fd[0], ...)` para ler do *pipe*, ficando bloqueado à espera se o processo "filho" ainda não tiver efectuado a operação de escrita.

```
/* Exemplo de utilizacao do mecanismo de comunicacao "pipe" (2 processos
   dependentes hierarquicamente (fork)). -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
  pid_t pid;
  int fd[2];
  int n;
```

```
  char buf[30];
  char msg[]="Filho comunica com pai";

  if (pipe(fd) != 0) {
    printf("Erro a criar pipe\n"); exit(-1);
  }

  if ((pid = fork()) == -1) {
    printf("Erro a criar processo\n"); exit(-1);
  }
  if (pid == 0) {
    printf("Ola'! Eu sou o filho. O meu pid=%d\n", getpid());
    printf("O filho vai enviar mensagem para o pai\n");
    write(fd[1], msg, sizeof(msg));
    exit(0);
  }
  else {
    printf("Eu sou o pai. O filho foi criado com pid=%d\n", pid);
    printf("Pai: vou esperar mensagem\n");
    n = read(fd[0], buf, sizeof(buf));
    printf("Pai: recebi msg. dim=%d info: %s \n", n, buf);
  }

  return 0;
}
```

4.4 IPC Sys V

Uma forma de conseguir a comunicação entre processos independentes é através da utilização dos mecanismos de comunicação entre processos (IPC) que foram desenvolvidos no âmbito do Unix Sistema V. Estes mecanismos utilizam um nome global (chave) que é conhecida à partida pelos vários processos interessados na sua utilização. Com base nessa *chave* é possível fazer a criação ou associação ao mecanismo de comunicação comum. São os seguintes os mecanismos suportados: mensagens (caixas-do-correio); memória partilhada; e semáforos.

4.5 Exemplos com mensagens IPC Sys V

Nos 2 programas seguintes ilustra-se um exemplo de uma aplicação cliente / servidor utilizando caixas-do-correio (mensagens IPC SysV).

O primeiro programa corresponde ao servidor. O servidor cria uma *caixa-do-correio* para receber mensagens com a função `msgget`, especificando como chave `SERVKEY` (definida como tendo valor 1), e indicando que deve ser criada caso ainda não exista (`IPC_CREAT`) e acessível em leitura/escrita para todos (`0666`). Caso essa operação seja efectuada com sucesso, será obtido um identificador (`msqids`) que permitirá efectuar outras operações sobre essa caixa-do-correio, como por exemplo, enviar ou receber mensagens. No caso do servidor, ele irá tentar receber mensagens dos eventuais clientes. Para isso chama a função `msgrcv`, especificando: o identificador da caixa-de-

correio `msqids`; a zona de memória onde a mensagem deve ser colocada; o tamanho máximo da mensagem; o tipo da mensagem (valor 0 significa qualquer tipo); flag para especificar comportamentos específicos, por exemplo, não bloquear (ver manual) (o valor 0 corresponde à situação por omissão, que implica bloqueio caso não exista nenhuma mensagem disponível).

O formato das mensagens deve obedecer aos seguintes requisitos. O primeiro campo da mensagem deverá obrigatoriamente ser um inteiro longo que deverá ter um valor positivo. Este campo é utilizado na recepção podendo permitr uma recepção selectiva. O resto da mensagem pode ter um formato arbitrário (vector de bytes). No caso do exemplo, existe um campo que conterá o identificador do cliente (valor que o cliente utilizou como chave para criar a sua caixa-do-correio), que irá ser utilizado pelo servidor para obter um identificador que lhe permita enviar mensagens para a caixa-do-correio do cliente.

Após a recepção de uma mensagem, o servidor:

- imprime o conteúdo da parte correspondente ao "texto";
- associa-se à caixa-de-correio do cliente `msgget` utilizando como chave o identificador que o cliente lhe enviou na mensagem (`msg.id`), para obter um identificador para essa caixa-do-correio `msqidc`). (O servidor apenas se associa, a caixa-de-correio é suposto existir, criada pelo cliente.);
- altera o texto da mensagem recebida, para a utilizar como mensagem de resposta;
- envia a mensagem de resposta usando `msgsnd` e utilizando o identificador obtido na associação anterior (`msqidc`).

Finalmente, o servidor elimina a sua caixa-de-correio com a função `msgctl`, e termina.

Nota: Os mecanismos de comunicação global mantém-se no sistema enquanto não forem eliminados. Se o programa não os eliminar poderá ser necessária a sua eliminação a partir da *shell* (ver comandos `ipcs` e `ipcrm`).

O segundo programa corresponde ao cliente. O cliente cria uma caixa-do-correio própria utilizando como chave o seu identificador de processo (pid). Seguidamente associa-se à caixa-do-correio do servidor utlizando a chave `SERVKEY` (a mesma que o servidor utilizou para criar a sua caixa-de-correio). A caixa-do-correio do servidor é suposto já existir. Para isso o programa do servidor deverá ter sido executado primeiro.

O cliente constrói uma mensagem especificando como tipo de mensagem o valor 1 (qualquer valor positivo servia), e colocando o seu identificador de processo (chave utilizada na criação da sua caixa-do-correio) na mensagem a enviar (campo `msg.id`). A mensagem, com o campo de "texto" preenchido com o caracter 'C', é depois enviada para a caixa-de-correio do servidor (`msgsnd(msqids, ...)`). Seguidamente o cliente fica à espera de uma resposta na sua caixa-de-correio (`msgrcv(msqidc, ...)`). Quando esta chega, imprime o "texto" da mensagem recebida, elimina a sua caixa-do-correio (`msgctl(msqidc, IPC_RMID, 0)`) e termina.

Código do servidor

```
/* Exemplo de utilizacao de mensagens IPC com 2 processos
   independentes hierarquicamente (Cliente, Servidor).
   Executar primeiro (em "background" ou noutra janela) o programa
   Servidor (este programa - "msgs"), e depois o programa
   Cliente ("msgc").  -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define TXTSIZE 10
#define SERVKEY 1

int main()
{
  int msqids, msqidc, i;
  struct {
    long    mtype;      /* message type */
    int     id;
    char    mtext[TXTSIZE];   /* message text */
  } msg;

  if ((msqids=msgget(SERVKEY, 0666|IPC_CREAT)) < 0) {
    perror("Servidor: Erro a criar queue servidor");
    exit(-1);
  }

  printf("Servidor vai receber\n");

  if (msgrcv(msqids, &msg, sizeof(msg.mtext)+sizeof(msg.id), 0, 0) < 0) {
    perror("Servidor: erro a receber mensagem");
  }
  else {
    for (i=0; i<TXTSIZE; i++)
      printf("%c", msg.mtext[i]);

    if ((msqidc=msgget(msg.id, 0)) < 0) {
      perror("Servidor: Erro a associar a queue cliente");
    }
    else {
      printf("Servidor vai enviar\n");
      for (i=0; i<TXTSIZE; i++)
msg.mtext[i] = 'S';
      if (msgsnd(msqidc, &msg, sizeof(msg.mtext)+sizeof(msg.id), 0) < 0) {
perror("Servidor: erro a enviar mensagem");
      }
    }
  }
  if (msgctl(msqids, IPC_RMID, 0) < 0) {
    perror("Servidor: Erro a eliminar queue servidor");
```

```
  }

  return 0;
}
```

Código do cliente

```
/* Exemplo de utilizacao de mensagens IPC com 2 processos
   independentes hierarquicamente (Cliente, Servidor).
   Executar primeiro (em "background" ou noutra janela) o programa
   Servidor ("msgs"), e depois o programa
   Cliente (este programa - "msgc").  -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define TXTSIZE 10
#define SERVKEY 1

int main()
{
  int msqids, msqidc, i;
  int cliid;
  struct {
    long    mtype;       /* message type */
    int     id;
    char    mtext[TXTSIZE];   /* message text */
  } msg;

  cliid = getpid();
  if ((msqidc=msgget(cliid, 0666|IPC_CREAT)) < 0) {
    perror("Cliente: Erro a criar queue cliente");
    exit(-1);
  }
  if ((msqids=msgget(SERVKEY, 0)) < 0) {
    perror("Cliente: Erro a associar a queue servidor");
    exit(-1);
  }

  printf("Cliente vai enviar\n");
  msg.mtype = 1;
  msg.id = cliid;
  for (i=0; i<TXTSIZE; i++)
    msg.mtext[i] = 'C';
  if (msgsnd(msqids, &msg, sizeof(msg.mtext)+sizeof(msg.id), 0) < 0) {
    perror("Cliente: erro a enviar mensagem");
  }
  else {
    printf("Cliente vai receber\n");
```

```
    if (msgrcv(msqidc, &msg, sizeof(msg.mtext)+sizeof(msg.id), 0, 0) < 0) {
      perror("Cliente: erro a receber mensagem");
    }
    else {
      for (i=0; i<TXTSIZE; i++)
printf("%c", msg.mtext[i]);
    }
  }
  if (msgctl(msqidc, IPC_RMID, 0) < 0) {
    perror("Cliente: Erro a eliminar queue cliente");
  }

  return 0;
}
```

4.6 Exemplos com memória partilhada IPC Sys V

A criação de uma zona de memória partilhada é feita de uma forma algo semelhante ao exemplo anterior. É necessária uma chave conhecida globalmente pelos vários processos intervenientes. Usando a função `shmget`, especificando essa chave como argumento, e incluindo a opção `IPC_CREAT`, para criar a zona de memória caso ela ainda não exista, obtem-se uma zona de memória com a dimensão especificada num outro argumento da função. Tal como anteriormente, a chamada à função `shmget` sem a opção `IPC_CREAT`, permite a associação a uma zona de memória já existente, obtendo um identificador para ela.

Para além da criação, ou associação, a uma zona de memória, é necessário "projectar" essa zona de memória no espaço de endereçamento do processo que a pretende utilizar. Ou seja, é necessário obter um apontador válido para essa zona de memória. Isso é feito utilizando a função `shmat`.

O exemplo representa uma aplicação produtor/consumidor. O produtor (primeiro programa) cria a zona de memória partilhada, à qual o consumidor se irá mais tarde também associar. A zona de memória é criada com a função `shmget`, utilizando a chave `MEMKEY` (definida neste exemplo com o valor 10), e especificando uma dimensão `SHMSIZE` (definida neste exemplo com o valor 20).

Após a criação com sucesso da zona de memória, é obtido um apontador para essa zona de memória com a função `shmat`, que projecta essa zona de memória no espaço de endereçamento do processo.

Com base nesse apontador, o processo produtor vai agora escrever na zona de memória partilhada (enche-a com o caracter 'P'). No fim de realizar a operação de escrita, "liberta" a zona de memória partilhada do seu espaço de endereçamento (com a função `shmdt`) e termina.

A zona de memória não é eliminada, pelo que se mantém no sistema, podendo ser acedida por outro processo que se lhe associe (neste caso o processo consumidor).

O processo consumidor (segundo programa) obtém um identificador para acesso à zona de memória partilhada usando também a função `shmget` (mas sem a opção de

criação IPC_CREAT) e especificando a mesma chave MEMKEY. No caso de sucesso (a zona de memória já existe e tem acesso a ela) o identificador obtido será usado para projectar essa zona de memória no seu espaço de endereçamento utilizando a função shmat (tal como o processo produtor havia feito). Com base no apontador obtido, vai agora aceder à zona de memória e imprimir o seu conteúdo (que tinha sido escrito pelo processo produtor).

Concluída a operação pretendida, vai retirar de novo a zona de memória do seu espaço de endereçamento (função shmdt), eliminar a zona de memória do sistema (função shmctl com a opção IPC_RMID), e terminar.

Neste exemplo não existe uma sincronização explícita entre o produtor e o consumidor. É assumido que o produtor é executado primeiro, e que só depois é executado o processo consumidor. Nesta situação particular não haverá conflitos. No entanto, numa situação mais genérica, com a possibilidade de haver mais do que um processo a aceder à mesma zona de memória, para garantir um acesso coerente, seria necessário utilizar mecanismos de sincronização explícita. Isso poderia ser feito recorrendo a semáforos do IPC Sys V (ver secção seguinte).

Código do produtor

```
/* Exemplo de utilizacao de memoria partilhada IPC com 2 processos
   independentes hierarquicamente (Produtor, Consumidor).
   Executar primeiro o Produtor (este programa - "shmp"), e depois o
   programa Consumidor ("shmc").  -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>

#define SHMSIZE 20
#define MEMKEY 10

int main()
{
  int shmid;
  char *shmaddr;
  int i;

  if ((shmid=shmget(MEMKEY, SHMSIZE, 0666|IPC_CREAT)) < 0) {
    perror("Produtor: Erro a criar memoria partilhada");
    exit(-1);
  }

  if ((shmaddr=shmat(shmid, NULL, 0)) == (void *) -1) {
    perror("Produtor: Erro a associar endereco");
    exit(-1);
  }
```

```
  printf("Produtor vai escrever\n");
  for (i=0; i<SHMSIZE; i++)
    shmaddr[i] = 'P';

  if (shmdt(shmaddr) < 0) {
    perror("Produtor: Erro a desassociar endereco");
  }

  return 0;
}
```

Código do consumidor

```
/* Exemplo de utilizacao de memoria partilhada IPC com 2 processos
   independentes hierarquicamente (Produtor, Consumidor).
   Executar primeiro o programa Produtor ("shmp"), e depois o
   programa Consumidor (este programa - "shmc").  -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>

#define SHMSIZE 20
#define MEMKEY 10

int main()
{
  int shmid;
  char * shmaddr;
  int i;

  if ((shmid=shmget(MEMKEY, SHMSIZE, 0666)) < 0) {
    perror("Consumidor: Erro a criar memoria partilhada");
    exit(-1);
  }

  if ((shmaddr=shmat(shmid, NULL, 0)) == (void *) -1) {
    perror("Consumidor: Erro a associar endereco");
    exit(-1);
  }

  printf("Consumidor vai ler\n");
  for (i=0; i<SHMSIZE; i++)
    printf("%c", shmaddr[i]);

  if (shmdt(shmaddr) < 0) {
    perror("Consumidor: Erro a desassociar endereco");
  }
  if (shmctl(shmid, IPC_RMID, 0) < 0) {
    perror("Consumidor: Erro a eliminar memoria");
```

```
  }

  return 0;
}
```

4.7 Exemplos com semáforos IPC Sys V

Nas concretizações mais comuns dos semáforos Posix (apresentados atrás), estes apenas são válidos dentro de um mesmo espaço de endereçamento (threads). Para usar semáforos entre processos, com espaços de endereçamento distintos, é necessário utilizar os semáforos IPC System V. Tal como nos mecanismos anteriores é utilizada uma chave como nome global para permitir a sua criação e utilização por parte dos processos.

No exemplo seguinte mostra-se uma aplicação constituída por 2 processos distintos, criados de forma independente, e que se sincronizam utilizando um semáforo. Mais concretamente, temos uma situação em que: o primeiro processo cria um semáforo com 0 unidades iniciais; ficando depois bloqueado à espera nesse semáforo, até que o outro processo o desbloquei.

O processo 1 (que deve ser executado primeiro porque é ele que vai criar e inicializar o semáforo) cria um semáforo utilizando a função `semget`, especificando como chave `PROC1KEY` (definida neste exemplo com o valor 11) e utilizando a opção `IPC_CREAT` para que o semáforo seja criado caso ainda não exista. O outro argumento, que neste caso toma o valor 1, indica que apenas se pretende criar um único semáforo no conjunto de semáforos identificado pela chave especificada (é possível criar vários semáforos num mesmo conjunto para facilitar operações envolvendo mais do que um semáforo – ver manual para mais pormenores).

Após a criação com sucesso do semáforo (conjunto de semáforos) obtém-se um identificador (`semid`, neste exemplo) que permite efectuar diversas operações sobre o semáforo criado. Essas operações podem ser quer de controlo (função `semctl`), quer operações normais de incrementar ou decrementar unidades do semáforo (função `semop`). Neste caso, o processo vai começar por realizar uma operação de controlo, utilizando a função `semctl` para inicializar o semáforo com 0 unidades. Para além do identificador do conjunto de semáforos (`semid`), são especificados: o índice do semáforo pretendido (0, neste caso em que existe só 1 semáforo); o código do comando a executar (`SETVAL`); e o argumento com o novo valor (a estrutura deste argumento é dependente do comando que se está a utilizar – ver manual).

Seguidamente o processo vai efectuar uma operação de decrementar uma unidade ao semáforo (*esperar*), utilizando a função `semop`. É possível realizar várias operações de forma atómica sobre o conjunto de semáforos. As várias operações são especificadas na forma de um vector de estruturas, em que cada estrutura identifica uma operação. Assim, nos argumentos para a função `semop`, é passado o apontador para o vector de estruturas, e o seu número (no caso de exemplo, esse valor é 1). Os campos dessa estrutura são os seguintes:

`sem_num` – para indicar o índice do semáforo;

`sem_op` – operação concreta: valor positivo, incrementar desse valor; valor negativo, decrementar; valor igual a 0, esperar até que o contador do semáforo seja 0;

`sem_flg` – possibilidade de especificar: `IPC_NOWAIT`, para apenas realizar as operações caso seja possível efectuá-las de imediato; e `SEM_UNDO`, para permitir reverter as operações efectuadas pelo processo, quando este terminar.

Como o semáforo tinha 0 unidades, se o outro processo ainda não tiver entretanto efectuado a operação de incrementar (assinalar), este processo irá ficar bloqueado no semáforo. Quando for desbloqueado, elimina o semáforo (`semctl(semid, 0, IPC_RMID)`) e termina.

Relativamente ao processo 2, este obtém um identificador para o semáforo utilizando a função `semget` com a mesma chave (`PROC1KEY`) e sem especificar que pretende criar o semáforo (é suposto já existir, criado pelo processo 1). Em caso de sucesso, com a função `semop`, realiza uma operação de incrementar uma unidade no semáforo (assinalar) e termina. O envio dessa unidade para o semáforo irá permitir desbloquear o processo 1 caso este já se encontre bloqueado. Se o processo 1 ainda estiver bloqueado no semáforo, o semáforo irá ficar com uma unidade, o que fará com que quando o processo 1 realizar a operação de "esperar", não se bloquei e possa avançar de imediato. De qualquer forma, o processo 1 só passará este ponto após o processo 2 ter incrementado o semáforo.

Código do processo 1

```
/* Exemplo de utilizacao de semaforos IPC com 2 processos
   independentes hierarquicamente.
   Executar primeiro (em "background" ou noutra janela) este
   programa ("sem1"), e depois o programa ("sem2").  -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

#define PROC1KEY 11

int main()
{
  int semid;
  union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
  } arg;
  struct sembuf sops[1];

  if ((semid=semget(PROC1KEY, 1, 0666|IPC_CREAT)) < 0) {
```

```
    perror("PROC1: Erro a criar semaforo");
    exit(-1);
  }

  arg.val = 0;
  if (semctl(semid, 0, SETVAL, arg) < 0) {
    perror("PROC1: Erro a inicializar semaforo");
    exit(-1);
  }

  sops[0].sem_num = 0;
  sops[0].sem_op  = -1;
  sops[0].sem_flg = 0;

  printf("PROC1 vai bloquear\n");
  if (semop(semid, sops, 1) < 0) {
    perror("PROC1: Erro a esperar semaforo");
  }
  printf("PROC1 desbloqueado\n");

  if (semctl(semid, 0, IPC_RMID) < 0) {
    perror("Erro a eliminar semaforo\n");
  }

  return 0;
}
```

Código do processo 2

```
/* Exemplo de utilizacao de semaforos IPC com 2 processos
   independentes hierarquicamente.
   Executar primeiro (em "background" ou noutra janela) o programa "sem1",
   e depois este ("sem2").  -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

#define PROC1KEY 11

int main()
{
  int semid;
  struct sembuf sops[1];

  if ((semid=semget(PROC1KEY, 1, 0666)) < 0) {
    perror("PROC2: Erro a obter semaforo");
    exit(-1);
  }

  sops[0].sem_num = 0;
```

```
  sops[0].sem_op  = 1;
  sops[0].sem_flg = 0;

  printf("PROC2 vai desbloquear PROC1\n");
  if (semop(semid, sops, 1) < 0) {
    perror("Erro a assinalar semaforo\n");
  }

  return 0;
}
```

Processos dependentes hierarquicamente

Os mecanismos IPC System V podem também ser utilizados por processos dependentes hierarquicamente (pai e filho). Nesse caso, não é necessário usar uma chave global, uma vez que os identificadores podem ser passados de "pai" para "filho" por "herança". Nesses casos, no lugar da chave, pode ser usado um valor pre-definido (`IPC_PRIVATE`).

No exemplo seguinte apresenta-se uma aplicação equivalente à anterior, mas agora com apenas 1 programa em que os 2 processos são obtidos através da utilização da função `fork`.

```
/* Exemplo de utilizacao de semaforos IPC com 2 processos
   dependentes hierarquicamente (fork). -- CRA
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

int main()
{
  int semid;
  union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
  } arg;
  struct sembuf sops[1];

  if ((semid=semget(IPC_PRIVATE, 1, 0666|IPC_CREAT)) < 0) {
    perror("Erro a criar semaforo");
    exit(-1);
  }

  arg.val = 0;
  if (semctl(semid, 0, SETVAL, arg) < 0) {
    perror("Erro a inicializar semaforo");
    exit(-1);
```

```
  }

  if (fork()==0) { /* filho */
    printf("Ola', eu sou o filho, pid=%d\n", getpid());
    sops[0].sem_num = 0;
    sops[0].sem_op  = -1;
    sops[0].sem_flg = 0;

    printf("Filho vai bloquear\n");
    if (semop(semid, sops, 1) < 0) {
      perror("Erro a esperar semaforo");
    }
    printf("Filho desbloqueado\n");
  }
  else { /* pai */
    printf("Ola', eu sou o pai, pid=%d\n", getpid());
    sops[0].sem_num = 0;
    sops[0].sem_op  = 1;
    sops[0].sem_flg = 0;

    sleep(5);
    printf("Pai vai desbloquear filho\n");
    if (semop(semid, sops, 1) < 0) {
      perror("Erro a assinalar semaforo\n");
    }
    printf("Pai continua. Vai dormir.\n");
    sleep(5);
    if (semctl(semid, 0, IPC_RMID) < 0) {
      perror("Erro a eliminar semaforo\n");
    }
  }

  return 0;
}
```

4.8 Exemplos com sockets

Os *sockets* constituem um outro exemplo de mecanismo de comunicação entre processos. Foram desenvolvidos no contexto do Unix de Berkeley (BSD), tendo tido uma grande divulgação com a expansão da Internet, a que estão associados. Um *socket* é, essencialmente, um ponto-terminal (end-point) para permitir efectuar as operações de envio e recepção (ou leitura / escrita). O mesmo socket é utilizado tanto para envio como recepção.

Existem vários domínios e vários protocolos que podem ser associados a um socket (ver manual para informação mais pormenorizada). Os domínios mais comuns são os correspondentes ao Unix e à Internet (com as variantes IPv4 e IPv6).

Existem também vários tipos de socket no que diz respeito ao tipo de comunicação (semântica) suportada. Os mais comuns são os sockets *datagrama* e os sockets *stream*. No primeiro caso é oferecido um serviço de comunicação por datagramas: mensagens com um dado comprimento máximo; sem ligação; e sem garantia de fiabilidade. No

caso dos sockets internet, está associado ao protocolo UDP (user datagram protocol). No caso dos sockets *stream* existe uma comunicação orientada à ligação, com garantia de sequenciação e garantia de fiabilidade (com retransmissões caso seja necessário) – associado ao protocolo TCP nos sockets internet.

Existe a possibilidade de associar um endereço/nome a um socket, de forma a este poder ser conhecido globalmente. Esse endereço toma a forma de um nome (cadeia de caracteres (string)), equivalente a um nome de ficheiro no caso dos sockets Unix, ou a forma de um endereço IP mais um porto, no caso dos sockets Internet.

Nota: ... internet ordem representação das variáveis bigendian

Existem várias funções associadas aos sockets. As mais representativas são:

`socket` – permite criar um socket especificando o domínio e o tipo;

`bind` – permite associar um endereço (nome ou endereço IP) ao socket;

`sendto` – para enviar um datagrama (sockets datagram);

`recvfrom` – para receber um datagrama (sockets datagram);

`listen` – para especificar que o socket vai ser utilizado para aceitar ligações, e o tamanho da fila de espera associada aos pedidos de ligação (sockets stream);

`accept` – para aceitar (servidor) o estabelecimento de uma ligação (sockets stream). Obtém-se um novo socket para uma comunicação de um-para-um;

`connect` – para pedir (cliente) o estabelecimento de uma ligação (sockets stream);

`read` – ler de um socket stream;

`write` – escrever num socket stream;

4.8.1 Sockets datagrama

Nesta secção apresenta-se um exemplo de uma aplicação cliente/servidor construída com base em sockets Unix datagrama.

No primeiro programa, correspondente ao Servidor, começa-se por criar um socket unix datagrama (`socket(PF_UNIX, SOCK_DGRAM, 0)`). Depois associa-se-lhe um endereço (SERVNAME – "/tmp/SERV") com a função `bind`, para que o socket seja conhecido no sistema a nível global (tornar o Servidor conhecido para os clientes).

Após essas inicializações, o Servidor chama a função `recvfrom`, ficando à espera de receber uma mensagem de um eventual cliente. Caso receba uma mensagem, o endereço do cliente estará colocado na variável `from`, cujo endereço foi passado como argumento na chamada à função `recvfrom`. Seguidamente é enviada uma resposta com a função `sendto`, utilizando o endereço do cliente obtido anteriormente. Finalmente, fecha-se o socket, e elimina-se o nome global.

No segundo programa, correspondente ao Cliente, também se começa por criar um socket unix datagrama (`socket(PF_UNIX, SOCK_DGRAM, 0)`), e associar-lhe um endereço (CLINAME – "/tmp/CLI") com a função `bind`. No entanto, este endereço, ao contrário do que acontecia com o Servidor, não necessita de ser conhecido globalmente à partida. Será o próprio sistema que, aquando da comunicação, o tornará disponível ao Servidor para que este possa responder ao Cliente, como se referiu atrás.

Após a criação do socket, o Cliente vai enviar uma mensagem ao Servidor. Para isso preenche a estrutura `to` com a informação relativa ao endereço do socket do Servidor (`AF_UNIX` e `SERVNAME`) que tem de ser conhecido à partida. A mensagem é enviada com a função `sendto`. Depois de enviada a mensagem, fica à espera da resposta chamando a função `recvfrom`. Quando a recebe, imprime-a no ecrã, fecha o seu socket, elimina o nome que lhe tinha atribuído, e termina.

Nota: No caso de se pretender ter a possibilidade de existirem vários clientes simultaneamente, é necessário que o nome a utilizar para associar ao socket do cliente seja diferente de cliente para cliente. Para isso pode ser usado o identificador do processo (pid) na construção desse nome.

Código do servidor

```
/* Exemplo de utilizacao de sockets unix datagrama na comunicacao
   entre 2 processos independentes hierarquicamente (Cliente,
   Servidor).  Executar primeiro (em "background" ou noutra janela) o
   programa Servidor (este programa - "sdsrv"), e depois o programa
   Cliente ("sdcli").  -- CRA
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

#define SERVNAME "/tmp/SERV"
#define MSG "Servidor responde!!!"

int main()
{
  int sd;
  struct sockaddr_un my_addr;
  socklen_t addrlen;
  struct sockaddr_un from;
  socklen_t fromlen;
  char buf[100];

  if ((sd = socket(PF_UNIX, SOCK_DGRAM, 0)) < 0 ) {
    perror("Erro a criar socket"); exit(-1);
  }
```

```
  my_addr.sun_family = AF_UNIX;
  memset(my_addr.sun_path, 0, sizeof(my_addr.sun_path));
  strcpy(my_addr.sun_path, SERVNAME);
  addrlen = sizeof(my_addr.sun_family) + strlen(my_addr.sun_path);

  if (bind(sd, (struct sockaddr *)&my_addr, addrlen) < 0 ) {
    perror("Erro no bind"); exit(-1);
  }

  fromlen = sizeof(from);
  if (recvfrom(sd, buf, sizeof(buf), 0, (struct sockaddr *)&from,
       &fromlen) < 0) {
    perror("Erro no recvfrom");
  }
  else {
    printf("SERV: Recebi: %s\n", buf);
    if (sendto(sd, MSG, strlen(MSG)+1, 0, (struct sockaddr *)&from,
       fromlen) < 0) {
      perror("Erro no sendto");
    }
  }

  close(sd);
  unlink(SERVNAME);

  return 0;
}
```

Código do cliente

```
/* Exemplo de utilizacao de sockets unix datagrama na comunicacao
   entre 2 processos independentes hierarquicamente (Cliente,
   Servidor).  Executar primeiro (em "background" ou noutra janela) o
   programa Servidor ("sdsrv"), e depois o programa
   Cliente (este programa - "sdcli").  -- CRA
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

#define SERVNAME "/tmp/SERV"
#define CLINAME "/tmp/CLI"
#define MSG "Cliente pergunta?!"

int main()
{
  int sd;
  struct sockaddr_un my_addr;
  socklen_t addrlen;
```

```
  struct sockaddr_un to;
  socklen_t tolen;
  char buf[100];

  if ((sd = socket(PF_UNIX, SOCK_DGRAM, 0)) < 0 ) {
    perror("Erro a criar socket"); exit(-1);
  }

  my_addr.sun_family = AF_UNIX;
  memset(my_addr.sun_path, 0, sizeof(my_addr.sun_path));
  strcpy(my_addr.sun_path, CLINAME);
  addrlen = sizeof(my_addr.sun_family) + strlen(my_addr.sun_path);

  if (bind(sd, (struct sockaddr *)&my_addr, addrlen) < 0 ) {
    perror("Erro no bind"); exit(-1);
  }

  to.sun_family = AF_UNIX;
  memset(to.sun_path, 0, sizeof(to.sun_path));
  strcpy(to.sun_path, SERVNAME);
  tolen = sizeof(my_addr.sun_family) + strlen(to.sun_path);

  if (sendto(sd, MSG, strlen(MSG)+1, 0, (struct sockaddr *)&to,
     tolen) < 0) {
    perror("CLI: Erro no sendto");
  }
  else {
    if (recvfrom(sd, buf, sizeof(buf), 0, (struct sockaddr *)&to,
 &tolen) < 0) {
      perror("CLI: Erro no recvfrom");
    }
    else {
      printf("CLI: Recebi: %s\n", buf);
    }
  }

  close(sd);
  unlink(CLINAME);

  return 0;
}
```

4.8.2 Sockets com ligação (“stream”)

Nesta secção apresenta-se de novo um exemplo de uma aplicação cliente/servidor, semelhante à anterior, mas desta vez construída com base em sockets Unix *stream*.

O primeiro programa, correspondente ao Servidor, também começa por criar um socket. Mas desta vez é um socket unix stream (`socket(PF_UNIX, SOCK_STREAM, 0)`). Tal como anteriormente, associa-lhe um endereço (SERVNAME – ”/tmp/SERV”) com a função `bind`, para que o socket seja conhecido no sistema a nível global (tornar o Servidor conhecido para os clientes).

Seguidamente, utilizando a função `listen`, informa o sistema que o socket criado vai ser utilizado para aceitar pedidos de ligação, e especifica que o tamanho da fila de espera associada a esses pedidos terá uma dimensão 1. Após essa operação, chama a função `accept`, onde ficará bloqueado até que algum cliente peça o estabelecimento de uma ligação. Quando tal acontecer, retorna da função `accept` com a criação de um novo socket que vai ser utilizado para a comunicação de "um-para-um" com o cliente que estabeleceu a ligação. A recepção e envio de mensagens é efectuada com as funções `read` e `write` (as mesmas que são normalmente utilizadas para ler e escrever em ficheiros). No caso concreto da aplicação apresentada, o Servidor utliza a função `read` para ficar à espera de uma mensagem do Cliente. Quando isso acontece, imprime essa mensagem no ecrã e envia uma mensagem de resposta utilizando a função `write`.

Finalmente fecha os 2 sockets (o socket inicial, utilizado para aceitar ligações, e o socket criado aquando do estabelecimento da ligação), elimina o nome global do socket do servidor, e termina.

Nota: Quando se pretende que um servidor possa atender vários clientes simultaneamente, é comum o servidor criar um novo processo (com a função `fork`) a seguir ao retorno da função `accept`, para atender o cliente, ficando o processo original em ciclo, de novo à espera no `accept` o estabelecimento de uma nova ligação.

Relativamente ao segundo programa, correspondente ao código do Cliente, temos também a criação de um socket unix *stream*. No entanto, o cliente não necessita de associar um endereço ao seu socket (pode fazê-lo, mas não é obrigatório) porque o estabelecimento de uma ligação (com a função `connect`) fornece uma comunicação de "um-para-um", pelo que o servidor não precisa de especificar o endereço para quem a mensagem vai ser enviada (está implícito).

Após o estabelecimento da ligação, o Cliente envia uma mensagem com a função `write` e fica à espera da resposta chamando a função `read`. Quando essa resposta é recebida, imprime-a no ecrã, fecha o socket e termina.

Código do servidor

```
/* Exemplo de utilizacao de sockets unix stream na comunicacao
   entre 2 processos independentes hierarquicamente (Cliente,
   Servidor).  Executar primeiro (em "background" ou noutra janela) o
   programa Servidor (este programa - "sssrv"), e depois o programa
   Cliente ("sscli").  -- CRA
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

#define SERVNAME "/tmp/SERV"
#define MSG "Servidor responde!!!"
```

```
int main()
{
  int sd, s;
  struct sockaddr_un my_addr;
  socklen_t addrlen;
  struct sockaddr_un from;
  socklen_t fromlen;
  char buf[100];

  if ((sd = socket(PF_UNIX, SOCK_STREAM, 0)) < 0 ) {
    perror("Erro a criar socket"); exit(-1);
  }

  my_addr.sun_family = AF_UNIX;
  memset(my_addr.sun_path, 0, sizeof(my_addr.sun_path));
  strcpy(my_addr.sun_path, SERVNAME);
  addrlen = sizeof(my_addr.sun_family) + strlen(my_addr.sun_path);

  if (bind(sd, (struct sockaddr *)&my_addr, addrlen) < 0 ) {
    perror("Erro no bind"); exit(-1);
  }
  if (listen(sd, 1) < 0 ) {
    perror("Erro no listen"); exit(-1);
  }

  fromlen = sizeof(from);
  if ((s=accept(sd, (struct sockaddr *)&from,  &fromlen)) < 0 ) {
    perror("Erro no accept"); exit(-1);
  }


  if (read(s, buf, sizeof(buf)) < 0) {
    perror("Erro no read");
  }
  else {
    printf("SERV: Recebi: %s\n", buf);
    if (write(s, MSG, strlen(MSG)+1) < 0) {
      perror("Erro no write");
    }
  }
  close(s);

  close(sd);
  unlink(SERVNAME);

  return 0;
}
```

Código do cliente

```
/* Exemplo de utilizacao de sockets unix stream na comunicacao
   entre 2 processos independentes hierarquicamente (Cliente,
   Servidor).  Executar primeiro (em "background" ou noutra janela) o
```

```
   programa Servidor ("sssrv"), e depois o programa
   Cliente (este programa - "sscli").  -- CRA
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

#define SERVNAME "/tmp/SERV"
#define MSG "Cliente pergunta?!"

int main()
{
  int sd;
  struct sockaddr_un srv_addr;
  socklen_t addrlen;
  char buf[100];

  if ((sd = socket(PF_UNIX, SOCK_STREAM, 0)) < 0 ) {
    perror("Erro a criar socket"); exit(-1);
  }

  srv_addr.sun_family = AF_UNIX;
  memset(srv_addr.sun_path, 0, sizeof(srv_addr.sun_path));
  strcpy(srv_addr.sun_path, SERVNAME);
  addrlen = sizeof(srv_addr.sun_family) + strlen(srv_addr.sun_path);

  if (connect(sd, (struct sockaddr *)&srv_addr, addrlen) < 0 ) {
    perror("Erro no connect"); exit(-1);
  }

 if (write(sd, MSG, strlen(MSG)+1) < 0) {
    perror("CLI: Erro no write");
  }
  else {
    if (read(sd, buf, sizeof(buf)) < 0) {
      perror("CLI: Erro no read");
    }
    else {
      printf("CLI: Recebi: %s\n", buf);
    }
  }

  close(sd);

  return 0;
}
```

5 Ficheiros em memória – mmap

Para uma maior facilidade de programação, quando é necessário efectuar operações de leitura/escrita num ficheiro, é possível projectar ("mapear") esse ficheiro (ou uma parte dele) na memória, e aceder-lhe através de apontadores em vez da utilização das funções `read` e `write`. O resultado final ficará depois guardado no ficheiro.

Nota: A utilização de ficheiros "mapeados" em memória também pode ser usado como memória partilhada (interface Posix).

Ao contrário do que acontece quando se utiliza a função `write`, em que o ficheiro pode ir aumentando de tamanho, no acesso através da projecção em memória, a dimensão do ficheiro não é alterada. Assim, quando o ficheiro é criado, e tem dimensão 0, é necessário utilizar a função `ftruncate` para garantir o tamanho desejado.

A projecção ("mapeamento") em memória é efectuada utilizando a função `mmap`. É possível especificar qual o endereço virtual onde se gostaria de ter o ficheiro mapeado, ou deixar que seja o sistema operativo a atribuir um endereço disponível. Para além do descritor do ficheiro (previamente aberto) e do deslocamento (*offset*) a partir do qual se pretende mapear, são fornecidos os parâmetros relativos a: dimensão da zona de memória; protecções / tipo de acesso (leitura, escrita, execução); e se a zona de memória relativa ao ficheiro vai ser partilhada ou não com outros processos (`MAP_SHARED, MAP_PRIVATE`). Em caso de sucesso a função `mmap` retorna um apontador para o início da zona de memória obtida, que pode ser utlizado pelo programa como qualquer outro apontador.

Nota: Em algumas versões do sistema operativo pode ser necessário ajustar o tamanho da zona de memória e o deslocamento no ficheiro (*offset*) para um múltiplo da dimensão da página de memória (que pode ser obtida com `sysconf(_SC_PAGE_SIZE)` e que normalmente toma o valor 4096).

Após a conclusão das operações pretendidas, a zona de memória pode ser retirada do espaço de endereçamento do processo com a função `munmap`, e o ficheiro fechado com a função `close`, ficando as alterações salvaguardadas no ficheiro.

Nota: Nos casos em que o tamanho especificado para a zona de memória não corresponde a um múltiplo do tamanho da página, podem ocorrer algumas situações particulares "estranhas" (no entanto facilmente explicáveis atendendo ao modo como funciona a unidade de gestão de memória): escritas na memória para além da dimensão especificada, mas que ficam dentro da mesma página, não provocam a terminação do processo, mas esses valores não ficam no ficheiro; escritas para além da dimensão da página provocam a terminação do processo (violação do espaço de endereçamento).

No programa apresentado, para ilustração do uso do mapeamento de ficheiros em memória, começa-se por abrir o ficheiro `FICH` (definido como "FICHEIRO.DAT"), usando a função `open(FICH, O_RDWR|O_CREAT, 0666)`. O ficheiro será aberto para leitura/escrita (`O_RDWR`), e, caso ainda não exista, será criado (`O_CREAT`) com permissão de leitura/escrita para todos (`0666`). Em caso de sucesso, é chamada a função `ftruncate` para estabelecer a dimensão do ficheiro em `MSIZE` (neste caso 100). Isto

é necessário no caso em que o ficheiro ainda não existia, e que, portanto, ao ser criado ficou com a dimensão 0.

Seguidamente, utilizando a função `mmap`, projecta-se o ficheiro, a partir do seu início (deslocamento 0), com uma dimensão 100, no espaço de endereçamento do processo, deixando o sistema operativo escolher um endereço virtual que esteja disponível (especificação de `NULL`, como endereço). A zona de memória pode ser acedida tanto em leitura como escrita (`PROT_READ|PROT_WRITE`), e pode ser partilhada com outros processos (`MAP_SHARED`). Em caso de sucesso, obtém-se um apontador para o início da zona de memória (que corresponde ao início do ficheiro). Com base nesse apontador preenche-se a zona de memória com o caracter 'A'. Finalmente, "desmapea-se" a zona de memória com a função `munmap`, fecha-se o ficheiro e termina-se. O ficheiro ficará com um conteúdo igual a 100 caracteres 'A'.

```
/* Exemplo de utilizacao do "mmap" para mapeamento de um ficheiro
em memoria (ou criacao de uma zona de memoria partilhada) -- CRA
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <errno.h>

#define FICH "FICHEIRO.DAT"
#define MSIZE 100

int main()
{
  int i, mfd;
  char *pa;

  if ((mfd=open(FICH, O_RDWR|O_CREAT, 0666 )) < 0) {  /* abrir ficheiro */
    perror("Erro a criar ficheiro");
    exit(-1);
  }
  else {
    if (ftruncate(mfd, MSIZE) < 0) {                  /* definir tamanho */
      perror("Erro no ftruncate");
      exit(-1);
    }
  }
  /* mapear ficheiro */
  if ((pa=mmap(NULL, 100, PROT_READ|PROT_WRITE, MAP_SHARED,
       mfd, 0)) < (char *)0) {
    perror("Erro em mmap");
    exit(-1);
  }

  /* aceder ao ficheiro atraves da memoria */
  for (i=0; i<MSIZE; i++) pa[i]='A';
```

```
  munmap(pa, MSIZE);
  close(mfd);


  return 0;
}
```

6 Tratamento de eventos: Sinais

Os sinais (*signals*) permitem o tratamento de acontecimentos assíncronos. É feita uma associação de uma função (gestor do sinal – *signal handler*) a um determinado sinal. Quando o processo recebe esse sinal (causado pelo hardware, execução de uma instrução, ou enviado por outro processo, dependendo do sinal concreto), essa função irá ser executada, interrompendo a execução do processo no ponto onde ele se encontrava.

Este mecanismo permite o tratamento de determinadas situações particulares associadas a vários tipos de sinais, e também pode constituir uma forma de comunicação/sincronização entre processos, uma vez que alguns deles podem ser enviados por programa.

A associação da função ao sinal pode ser feita utilizando as funções `signal` ou `sigaction`. No primeiro caso (interface original) a função é especificada directamente como argumento. No segundo caso (interface definida pela norma POSIX) utiliza-se uma estrutura auxiliar que permite especificar para além da função, outra informação adicional (flags, máscara – ver manual).

Um sinal pode ser configurado para ter um gestor (*handler*) associado, ser ignorado (`SIG_IGN`), ou ter o tratamento por omissão (`SIG_DFL`). O tratamento por omissão para a maior parte dos sinais é a terminação do processo.

Nota: A configuração dos sinais `SIGKILL` e `SIGSTOP` não pode ser alterada. Quando recebidos implicam a terminação do processo.

Nota: Quando se utiliza a função `signal` para especificar o gestor do sinal, e dependendo do sistema unix que se está a utilizar, podem existir comportamentos distintos. No unix original (variante System V) a associação do gestor ao sinal é válida apenas uma vez, retornando em seguida ao tratamento por omissão (`SIG_DFL`). Para manter a associação é necessário chamar de novo a função `signal`. No caso do unix de berkeley (BSD) a associação é mantida. No Linux é possível configurar qual o comportamento que é seguido. Com a utilização da função `sigaction` (recomendada pela interface POSIX) é possível através da utilização das flags alterar o comportamento por omissão.

Nota: Quando um processo está bloqueado numa função do sistema, e o processo recebe um sinal, pode haver um retorno dessa função com a indicação da existência de erro (retorno -1) e código de erro `EINTR`. O programador deve estar atento a essa situação, consultar os manuais das funções utilizadas, e agir de acordo com a conveniência da aplicação concreta.

Existem várias funções relacionadas com a gestão dos sinais. Para uma descrição mais pormenorizada ver as aulas teóricas e os manuais das várias funções. De qualquer forma, em traços gerais, para além das funções já referidas acima para definir os gestores, temos: funções para o processo se bloquear em espera que ocorra um sinal (e.g. `pause`, `sigsuspend`); funções para enviar sinais (e.g. `kill`, `sigqueue`); funções para manipular máscaras e eventualmente bloquear sinais (e.g. `sigprocmask`).

Como se disse acima, um processo pode receber um sinal como consequência: de um problema de hardware; da execução de uma determinada instrução; ou enviado por outro processo. Um processo pode enviar um sinal utilizando a função `kill`, e especificando o identificador do processo (pid) a quem pretende enviar o sinal, e qual o sinal que pretende enviar. Um sinal também pode ser enviado a partir da linha de comando (shell) utilizando o comando `kill`, de funcionalidade equivalente à função referida atrás para utilização por programas.

Com a interface POSIX-RT foram introduzidas algumas extensões no tratamento dos sinais. Para um conjunto restrito de sinais (`SIGRTMIN -- SIGRTMAX`) é possível associar informação adicional, bem como ter uma fila de espera para sinais recebidos, ou esperar directamente que o sinal seja recebido sem ter a necessidade de especificar um gestor para esse sinal (função `sigwaitinfo`).

Os exemplos seguintes ilustram a utilização de sinais.

Exemplo-1: Utilização da função `signal` para definir um gestor para o sinal `SIGTERM`. O programa principal fica em ciclo, bloqueando-se com a função `pause`, à espera da recepção do sinal. O sinal pode ser enviado a partir da linha de comando (`shell`).

Exemplo-2: Situação semelhante à do exemplo anterior, mas utilizando a função `sigaction` para definir o gestor do sinal.

Exemplo-3: Exemplo com a utilização de 2 processos: "pai" e "filho". O "filho" define um gestor para o sinal `SIGUSR1` com a função `sigaction`, e fica à espera do sinal com a função `sigsuspend`, terminando quando recebe o sinal. O "pai" envia o sinal `SIGUSR1` ao "filho", utilizando a função `kill`, ficando seguidamente à espera que o "filho" termine (função `wait`) e terminado depois também. Este exemplo inclui ainda a manipulação de máscaras associadas aos sinais.

Exemplo-4: Exemplo com uma estrutura semelhante à do exemplo anterior, mas utilizando as extensões POSIX-RT (válidas para sinais com números entre `SIGRTMIN` e `SIGRTMAX`), que permitem a existência de mais informação associada ao sinal, desde que se utiliza a flag `SA_SIGINFO` na especificação do gestor. Parte dessa informação adicional pode ser especificada no momento do envio do sinal, desde que se use a função `sigqueue` em vez de `kill`.

Exemplo-5: Variação do exemplo anterior em que, em vez de se especificar um gestor para o sinal, se fica bloqueado na função `sigwaitinfo` à espera da recepção do sinal.

Exemplo 1 – signal

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>


void sighand(int sn)
{
  printf("sighand: sn=%d\n", sn);
  signal(SIGTERM, sighand);
}

int main()
{

  printf("main: signal\n");
  signal(SIGTERM, sighand);
  while(1){
    printf("main: pause\n");
    pause();
  }
  printf("main: exit\n");

  return 0;
}
```

Exemplo 2 – sigaction

```
/* #define _POSIX_SOURCE */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

struct sigaction sa;

void sighand(int sn)
{
  printf("sighand: sn=%d\n", sn);
}

int main()
{

  printf("main: sigaction\n");
  sa.sa_flags = 0;
  sa.sa_handler = sighand;
  sigaction(SIGTERM, &sa, NULL);
  while(1){
    printf("main: pause\n");
    pause();
  }
  printf("main: exit\n");
```

```
  return 0;
}
```

Exemplo 3 – kill

```
/* #define _POSIX_SOURCE */
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIG_STOP_CHILD SIGUSR1

int main()
{
pid_t pid;
sigset_t newmask, oldmask;

    if ((pid = fork()) == 0) {   /*Child*/
        struct sigaction action;
        void catchit();

        sigemptyset(&newmask);
        sigaddset(&newmask, SIG_STOP_CHILD);
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);

        action.sa_flags = 0;
        action.sa_handler = catchit;

        if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {
            perror("sigusr: sigaction");
            _exit(1);
        }
        sigsuspend(&oldmask);
    }
    else {                              /* Parent */
        int stat;
        sleep(1);
        kill(pid, SIG_STOP_CHILD);
        pid = wait(&stat);
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
    }
    return 0;
}
void catchit(int signo)
{
       printf("Signal %d received from parent\n", signo);
       _exit(0);
}
```

Exemplo 4 – Extensões POSIX-RT: sigqueue

```
/* #define _POSIX_C_SOURCE 199309L */ /*199506L 199309L*/
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIG_STOP_CHILD SIGRTMIN+1

int main()
{
pid_t pid;
sigset_t newmask, oldmask;

    if ((pid = fork()) == 0) {   /*Child*/
        struct sigaction action;
        void catchit();

        sigemptyset(&newmask);
        sigaddset(&newmask, SIG_STOP_CHILD);
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);

        action.sa_flags = SA_SIGINFO;
        action.sa_sigaction = catchit;

        if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {
            perror("sigusr: sigaction");
            _exit(1);
        }
        sigsuspend(&oldmask);
    }
    else {                              /* Parent */
        union sigval sval;
        int stat;
        sval.sival_int = 1;
        sleep(1);
        sigqueue(pid, SIG_STOP_CHILD, sval);
        pid = wait(&stat);
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
        _exit(0);
    }
    return 0;
}

void catchit(int signo, siginfo_t *info, void *extra)
{
  /*       void *ptr_val = info->si_value.sival_ptr; */
       int int_val = info->si_value.sival_int;
       printf("Signal %d, value %d  received from parent\n", signo, int_val);
       _exit(0);
}
```

Exemplo 5 – Extensões POSIX-RT: sigwaitinfo

```
/* #define _POSIX_C_SOURCE 199309L */ /*199506L 199309L*/
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIG_STOP_CHILD SIGRTMIN+1

int main()
{
  pid_t pid;
  sigset_t newmask;
  int rcvd_sig;
  siginfo_t info;

  if ((pid = fork()) == 0) {       /*Child*/

    sigemptyset(&newmask);
    sigaddset(&newmask, SIG_STOP_CHILD);
    sigprocmask(SIG_BLOCK, &newmask, NULL);

    while (1) {
      rcvd_sig = sigwaitinfo(&newmask, &info);
      if (rcvd_sig == -1) {
perror("sigusr: sigwaitinfo");
_exit(1);
      }
      else {
printf("Signal %d, value %d  received from parent\n",
       rcvd_sig, info.si_value.sival_int);
_exit(0);
      }
    }
  }
  else {                           /* Parent */
    union sigval sval;
    int stat;
    sval.sival_int = 1;
    sleep(1);
    sigqueue(pid, SIG_STOP_CHILD, sval);
    pid = wait(&stat);
    printf("Child exit status = %d\n", WEXITSTATUS(stat));
    _exit(0);
  }
  return 0;
}
```

7 Facilidades de tempo-real – Acesso ao relógio

Embora nem todas as aplicações tenham requisitos de tempo-real, algumas necessitam de ter alguma noção de tempo. Por exemplo, pode haver situações em que é necessário: medir a duração de um intervalo de tempo; esperar, adormecendo o processo, durante um determinado tempo; especificar um tempo máximo de espera por um dado acontecimento, para evitar um eventual bloqueio indefinido; estampilhar eventos com a data/hora de ocorrência; especificar alarmes e temporizadores; etc.

Para dar suporte às situações referidas, é necessário que o sistema possua um relógio, e que seja possível aceder a esse relógio. Embora o assunto não seja tratado de um forma exaustiva, nesta secção apresentam-se alguns exemplos de utilização de funções que permitem lidar com aspectos relacionados com o tempo. Nomeadamente, apresentam-se: funções e estruturas de dados usadas para aceder e manipular o tempo obtido a partir do relógio do sistema; exemplos de multiplexagem das operações de entrada/saída e especificação de esperas temporizadas (utilização da função `select`); exemplos de manipulação e conversão de formatos do tempo.

Protótipos e estruturas relacionadas com o tempo

```
time_t time( time_t *t );
int     gettimeofday( struct timeval *tv, struct timezone *tz );
int     clock_gettime( clockid_t clk_id, struct timespec *tp );
struct timeval {
     time_t       tv_sec;     /* seconds */
     suseconds_t tv_usec; /* microseconds */
};
struct timespec {
     time_t tv_sec;       /* seconds */
     long tv_nsec;        /* nanoseconds */
};
```

Exemplos de acesso ao relógio do sistema

```
//Example: Returning Time
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
int main()
{
     time_t t;
     struct timeval tv;
     struct timespec ts;
     // 3 ways to access system clock
     time( &t );
     gettimeofday( &tv, NULL) ;
     clock_gettime( CLOCK_REALTIME, &ts );
     printf("time returns:     %ld s\n", t);
```

```
    printf("gettimeofday returns: %ld s and %ld us\n", tv.tv_sec, tv.tv_usec);
    printf("clock_gettime returns: %ld s and %ld ns\n", ts.tv_sec, ts.tv_nsec);
    return 0;
}
```

Exemplos de conversão de formatos de tempo

```
//Example: Converting Time
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
int main() {
  time_t t;
  struct tm tm;
  char str[26];
  time(&t);                    // get time
  localtime_r(&t, &tm);        // convert time_t -> tm
            // convert tm -> str asci (e.g. "Tue Nov 15 15:45:25 WET 2011")
  asctime_r(&tm, &str[0]);
            // ctime(t) is equivalent to asctime(localtime(t))
  printf("ctime(t): %s\n", ctime(&t));
            // specify format of str
  strftime(&str[0], sizeof(str), "%d/%m/%Y %H:%M:%S\n", &tm);
  printf("str: %s\n", str);                 // e.g. "15/11/2011 15:45:25"
  printf("data/hora (dd/mm/yyyy hh:mm:ss):");
  fgets(&str[0], sizeof(str), stdin);
            // parse str according to specified format
  strptime(str, "%d/%m/%Y %H:%M:%S\n", &tm);
  t = mktime(&tm);             // convert tm -> time_t
}
```

Select: multiplexagem e espera temporizada

```
int select( int nfds, fd_set *readfds, fd_set *writefds,
          fd_set *errorfds, struct timeval *timeout );
void  FD_ZERO( fd_set *fdset );
void  FD_SET( int fd, fd_set *fdset );
void  FD_CLR( int fd, fd_set *fdset );
int   FD_ISSET( int fd, fd_set *fdset );
```

Exemplo de multiplexagem – Select

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
int main() {
  int p1[2], p2[2];
  int mx; fd_set rs;
  if (pipe(p1) != 0) { perror("pipe1"); return (-1); }
  if (pipe(p2) != 0) { perror("pipe2"); return (-1); }
```

```
  /* create other processes; finish setup for pipes */
  if (p1[0] > p2[0]) mx=p1[0]; else mx=p2[0]; // find biggest fd
  while (1) {
    FD_ZERO(&rs);
    FD_SET(p1[0], &rs);
    FD_SET(p2[0], &rs);
    if ((select(mx+1, &rs, NULL, NULL, NULL) == -1) &&
          (errno != EINTR)) perror ("Select failed");
    else {
      if (FD_ISSET(p1[0], &rs)) {
         // read and process pipe p1 input
      }
      if (FD_ISSET(p2[0], &rs)) {
         // read and process pipe p2 input
      }
    }
  }
}
```

Exemplo de espera temporizada – Select

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
  fd_set rfds;
  struct timeval tv;
  int retval;
  /* Watch stdin (fd 0) to see when it has input. */
  FD_ZERO(&rfds);
  FD_SET(0, &rfds);
  /* Wait up to five seconds. */
  tv.tv_sec = 5;
  tv.tv_usec = 0;
  retval = select(1, &rfds, NULL, NULL, &tv);
  if (retval)
    printf("Data is available now.\n");
  /* FD_ISSET(0, &rfds) will be true. */
  else
    printf("No data within five seconds.\n");
  return 0;
}
```

8 Interpretador de comandos – cmd

Ainda que no contexto do desenvolvimento de aplicações comerciais a interface com o utilizador possa ser um aspecto importante, no contexto desta disciplina o objectivo fundamental é dominar os aspectos relativos à programação sistema e à concorrência. Dessa forma, no projecto de laboratório para avaliação, opta-se por simplificar a parte

relativa à criação da interface com o utilizador, e fornecer código (baseado em código existente, ao qual foram feitas algumas adaptações), correspondente a um interpretador de comandos, que permite construir com relativa facilidade a interface utilizador desejada.

Este interpretador de comandos (`cmd`), cujos ficheiros de código se apresentam a seguir, possui um esquema de funcionamento simples. Existe uma tabela onde são incluídos os nomes dos vários comandos desejados, e se faz corresponder a cada comando uma função. Durante a execução é feita a descodificação dos comandos (*parsing*) e chamada a função que lhe está associada para executar o comando.

Ficheiro makefile

```
CC=gcc
CFLAGS= -Wall -g

OBJS= cmd.o comando.o monitor.o

cmd: $(OBJS)
$(CC) -o cmd $(OBJS)

clean:
rm cmd $(OBJS)
```

Ficheiro cmd.c

```
#include <stdio.h>

extern void monitor();

int main()
{
  monitor();

  return 0;
}
```

Ficheiro monitor.c

```
/*************************************************************************
| File: monitor.c
|
| Autor: Carlos Almeida (IST), from work by Jose Rufino (IST/INESC),
|        from an original by Leendert Van Doorn
| Data:  Nov 2002
*************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <ctype.h>

/*-------------------------------------------------------------------------+
| Headers of command functions
+-------------------------------------------------------------------------*/
extern void cmd_sair (int, char** );
extern void cmd_test (int, char** );
       void cmd_sos  (int, char** );

/*-------------------------------------------------------------------------+
| Variable and constants definition
+-------------------------------------------------------------------------*/
const char TitleMsg[] = "\n Application Control Monitor\n";
const char InvalMsg[] = "\nInvalid command!";

struct  command_d {
  void  (*cmd_fnct)();
  char* cmd_name;
  char* cmd_help;
} const commands[] = {
  {cmd_sos,  "sos","                  help"},
  {cmd_sair, "sair","                 sair"},
  {cmd_test, "teste","<arg1> <arg2>   comando de teste"}
};

#define NCOMMANDS  (sizeof(commands)/sizeof(struct command_d))
#define ARGVECSIZE 3
#define MAX_LINE   50

/*-------------------------------------------------------------------------+
| Function: cmd_sos - provides a rudimentary help
+-------------------------------------------------------------------------*/
void cmd_sos (int argc, char **argv)
{
  int i;

  printf("%s\n", TitleMsg);
  for (i=0; i<NCOMMANDS; i++)
    printf("%s %s\n", commands[i].cmd_name, commands[i].cmd_help);
}

/*-------------------------------------------------------------------------+
| Function: my_getline        (called from monitor)
+-------------------------------------------------------------------------*/
int my_getline (char** argv, int argvsize)
{
  static char line[MAX_LINE];
  char *p;
  int argc;

  fgets(line, MAX_LINE, stdin);

  /* Break command line into an o.s. like argument vector,
     i.e. compliant with the (int argc, char **argv) specification --------*/

  for (argc=0,p=line; (*line != '\0') && (argc < argvsize); p=NULL,argc++) {
```

```
    p = strtok(p, " \t\n");
    argv[argc] = p;
    if (p == NULL) return argc;
  }
  argv[argc] = p;
  return argc;
}

/*-------------------------------------------------------------------------+
| Function: monitor        (called from main)
+-------------------------------------------------------------------------*/
void monitor ()
{
  static char *argv[ARGVECSIZE+1], *p;
  int argc, i;

  printf("%s Type sos for help\n", TitleMsg);
  for (;;) {
    printf("\nCmd> ");
    /* Reading and parsing command line  ---------------------------------*/
    if ((argc = my_getline(argv, ARGVECSIZE)) > 0) {
      for (p=argv[0]; *p != '\0'; *p=tolower(*p), p++);
      for (i = 0; i < NCOMMANDS; i++)
if (strcmp(argv[0], commands[i].cmd_name) == 0)
  break;
      /* Executing commands ------------------------------------------------*/
      if (i < NCOMMANDS)
commands[i].cmd_fnct (argc, argv);
      else
printf("%s", InvalMsg);
    } /* if my_getline */
  } /* forever */
}
```

Ficheiro comando.c

```
/*************************************************************************
| File: comando.c  -  Concretizacao de comandos (exemplo)
|
| Autor: Carlos Almeida (IST)
| Data:  Nov 2002
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>

/*-------------------------------------------------------------------------+
| Function: cmd_sair - termina a aplicacao
+-------------------------------------------------------------------------*/
void cmd_sair (int argc, char **argv)
{
  exit(0);
}

/*-------------------------------------------------------------------------+
| Function: cmd_test - apenas como exemplo
```

```
+-----------------------------------------------------------------------*/
void cmd_test (int argc, char** argv)
{
  int i;

  /* exemplo -- escreve argumentos */
  for (i=0; i<argc; i++)
    printf ("\nargv[%d] = %s", i, argv[i]);
}
```

9 Desenvolvimento dos Trabalhos e Avaliação

O objectivo fundamental das aulas de laboratório e respectivos trabalhos/projectos é incentivar os alunos a praticar, aprofundar e consolidar os conhecimentos adquiridos. No contexto desta disciplina, reveste-se de particular importância a interacção com o sistema operativo e a compreensão dos mecanismos fundamentais associados ao desenvolvimento de aplicações concorrentes.

Os alunos são incentivados a desenvolver um trabalho contínuo ao longo do semestre. Numa primeira fase, começam por se familiarizar com exemplos de programas fornecidos pelo docente (apresentados neste documento). Para além de tentarem perceber o seu funcionamento (lendo, compilando, executando), é-lhes pedido/sugerido que efectuem alterações a esses exemplos de forma a aumentar a sua funcionalidade, e/ou melhor porem em evidência determinados aspectos referidos na matéria teórica. Alguns exemplos de alterações sugeridas são: verificação dos espaços de endereçamento dos processos e das threads; colocar os servidores em ciclo, e permitir vários clientes; comunicação bidireccional; sincronização; etc.

9.1 Metas intercalares / fases de avaliação

Se bem que todo o trabalho desenvolvido pelos alunos ao longo do semestre vai permitindo um melhor conhecimento por parte do docente das capacidades e conhecimentos adquiridos pelos alunos, é importante existirem pontos concretos de avaliação, de forma a garantir um acompanhamento adequado das matérias, e evitar potênciais "desligamentos" e/ou concentração de todo o trabalho na fase final do semestre (o que normalmente acarreta maus resultados). Assim, existem metas intercalares.

A primeira meta intercalar consiste num pequeno trabalho desenvolvido numa aula de laboratório. Podendo ter como base alguns dos exemplos fornecidos, é pedida a construção de uma aplicação simples com comunicação e sincronização, e envolvendo processos, threads, pipes e semáforos posix.

A segunda meta intercalar já faz parte do projecto. Tem por objectivo garantir um correcto faseamento e evitar tendências de adiamentos. É pedido que exista, ainda que com funcionalidade parcial/reduzida, a estrutura base da aplicação com um subconjunto dos seus componentes finais.

O projecto será depois entregue e visualizado, numa aula de laboratório (tipicamente na penúltima semana de aulas), para averiguar da sua funcionalidade.

Finalmente serão realizadas discussões (prova oral), na última semana de aulas, que irão permitir: uma análise mais pormenorizada das soluções apresentadas; a identificação de eventuais erros de concepção; e a atribuição de notas aos vários elementos do grupo.

9.2 Projecto

O projecto constitui o trabalho de laboratório principal. Consiste essencialmente numa aplicação concorrente, em que são manipulados vários objectos sistema, tais como processos e *threads*, e são utilizados vários mecanismos de comunicação e sincronização. De uma maneira geral, procura-se que a aplicação cubra um conjunto diversicado de aspectos, exigindo-se a utilização de alguns mecanismos específicos em determinadas situações.

Os projectos diferem de ano para ano, mantendo, no entanto, uma estrutura base semelhante. A complexidade e o tipo de problemas a resolver está condicionada ao tempo disponível e a uma carga aceitável. Para facilitar essa gestão, opta-se por simplificar determinados aspectos considerados menos relevantes, tais como a interface do utlizador, bem como o fornecimento de alguns exemplos de código. Em anexo é apresentado um enunciado de projecto.

10 Perspectivas de Evolução Futura

Embora já seja coberto na disciplina um conjunto significativo de aspectos, existem outros que também poderiam ser abordados. No entanto, como o tempo é limitado, isso poderá implicar uma troca ou simplificação de alguns dos assuntos abordados actualmente, de forma a poder contemplar outros.

De qualquer forma, alguns aspectos que poderia ser interessante considerar são:

- melhor domínio de ambientes integrados (e.g. eclipse);
- controlo de versões;
- shell scripts;
- criação e gestão de bibliotecas (*libraries*);
- interfaces gráficas simples (e.g. uGui);
- maior interacção com dispositivos periféricos;
- pequenos trabalhos usando outros sistemas operativos (e.g. núcleos multitarefa, windows).

11 Bibliografia

A realização dos trabalhos de laboratório pressupõe o domínio dos conceitos fundamentais relacionados com os sistemas operativos, tanto de um ponto-de-vista mais teórico, como de um ponto-de-vista mais prático com o conhecimento da sintaxe das várias funções que constituem a interface disponível. Assim a bibliografia a consultar é constituída por:

- livros sobre sistemas operativos, e alguns casos particulares do Unix (alguns exemplos em baixo);
- informação disponibilizada nas aulas teóricas e de laboratório;
- acetatos;
- exemplos de programas fornecidos;
- páginas dos manuais das funções.

Alguns livros

- Modern Operating Systems (second edition), Tanenbaum, Andrew S., 2001, Prentice-Hall
- Advanced Programming in the UNIX Environment, 2nd ed., W. Richard Stevens, Stephen A. Rago, 2005, Addison-Wesley
- Operating System Concepts (fifth edition), Silberchatz, Abraham, & Galvin, Peter, 1997., John Wiley & Sons, Inc.
- Sistemas Operativos, J.A. Marques, P. Ferreira, C. Ribeiro, L. Veiga, R. Rodrigues., 2009., FCA - Editora de Informática.
- Real-Time Systems and Programming Languages (third edition)., Burns, A., & Wellings, A., 2001., Addison-Wesley Publishers Ltd.

A Enunciado do Projecto

Neste anexo apresenta-se um exemplo de enunciado de projecto. Embora todos os anos o enunciado seja diferente, a sua estrutura é semelhante, tentando, na medida do possível, fazer com que os alunos abordem um conjunto significativo, e variado, dos vários aspectos relacionados com a programação de sistemas computacionais.