
Process Management

Processes, Threads and Multithreading, Process
Switching & Process Scheduling

PEDRO MARTINS

February 1, 2018

Contents

1	Processes and Threads	4
1.1	Arquitetura típica de um computador	4
1.2	Programa vs Processo	4
1.3	Execução num ambiente multiprogramado	5
1.4	Modelo de Processos	5
1.5	Diagrama de Estados de um Processo	6
1.5.1	Swap Area	8
1.5.2	Temporalidade na vida dos processos	9
1.6	State Diagram of a Unix Process	11
1.7	Supervisor preempting	12
1.8	Unix – traditional login	12
1.9	Criação de Processos	13
1.10	Execução de um programa em C/C++	17
1.11	Argumentos passados pela linha de comandos e variáveis de ambiente	17
1.12	Espaço de Endereçamento de um Processo em Linux	18
1.12.1	Process Control Table	19
2	Threads	20
2.1	Diagrama de Estados de uma thread	22
2.2	Vantagens de Multithreading	22
2.3	Estrutura de um programa multithreaded	23
2.4	Implementação de Multithreading	23
2.4.1	Libreria pthread	24
2.5	Threads em Linux	25
3	Process Switching	26
3.1	Exception Handling	28
3.2	Processing a process switching	29
4	Processor Scheduling	29
4.1	Scheduler	30
4.1.1	Long-Term Scheduling	30
4.1.2	Medium Term Scheduling	30
4.1.3	Short-Term Scheduling	31
4.2	Critérios de Scheduling	31
4.2.1	User oriented	31
4.2.2	System oriented	32
4.3	Preemption & Non-Preemption	32
4.4	Scheduling	33
4.4.1	Favouring Fearness	33
4.4.2	Priorities	34
	Prioridades Estáticas	34

	Prioridades Dinâmicas	35
	Shortest job first (SJF) / Shortest process next (SPN)	36
4.5	Scheduling Policies	37
4.5.1	First Come, First Serve (FCFS)	37
4.5.2	Round-Robin	37
4.5.3	Shortest Process Next (SPN) ou Shortest Job First (SJF)	38
4.5.4	Linux	38
	Algoritmo Tradicional	39
4.6	Novo Algoritmo	39

1 Processes and Threads

1.1 Arquitectura típica de um computador

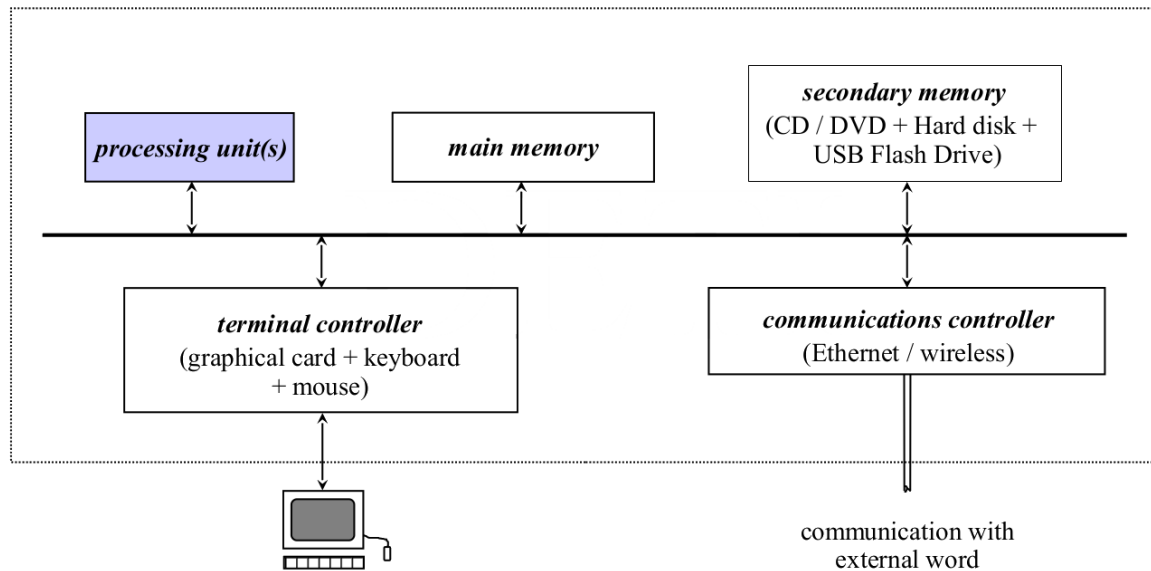


Figure 1: Arquitectura típica de um computador

1.2 Programa vs Processo

- **programa:** conjunto de instruções que definem como uma tarefa é executada num computador
 - É apenas um **conjunto de instruções** (código máquina), nada mais
 - Para realizar essas **funções/instruções/tarefas** o código (ou a versão compilada dele) tem de ser executado(a)
- **processo:** Entidade que representa a **execução de um programa**
 - Representa a sua atividade
 - Tem associado a si:
 - * código que ao contrário do programa está armazenado num endereço de memória (addressing space)
 - * **dados** (valores das diferentes variáveis) da execução corrente
 - * valores atuais dos registos internos do processador
 - * dados dos I/Os, ou seja, dados que estão a ser transferidos entre dispositivos de input e output
 - * Estado da execução do programa, ou seja, qual a próxima execução a ser executada (registo PC)
 - Podem existir diferentes processos do mesmo programa
 - * Ambiente **multiprogramado** - mais processos que processadores

1.3 Execução num ambiente multiprogramado

O sistema assume que o processo que está na posse do processador irá **ser interrompido**, podendo assim executar outro processo e dar a “sensação” em **macro tempo** de **simultaneidade**. Nestas situações, o OS é responsável por:

- tratar da **mudança do contexto de execução**, guardando
 - o valor dos registos internos
 - o valor das variáveis
 - o endereço da próxima instrução a ser executada
- chamar o novo processo que vai ocupar agora o CPU e:
 - Esperar que o novo processo termine a realização das suas operações **ou**
 - Interromper o processo, **parando a sua execução no** processador quando este esgotar o seu **time quantum**

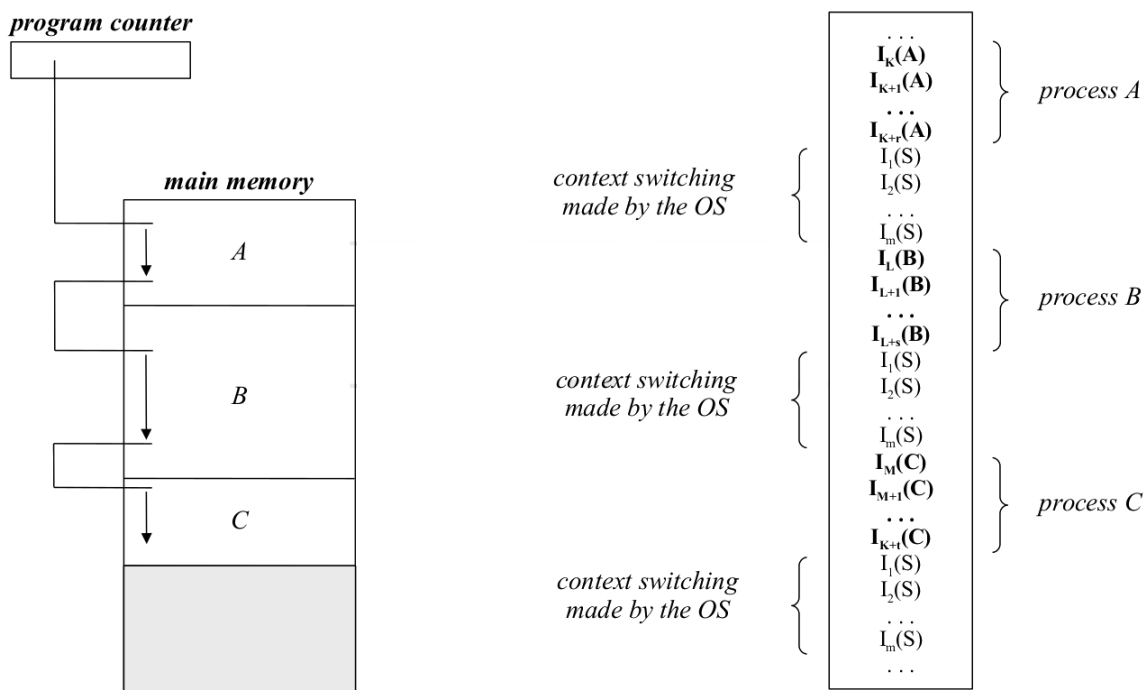


Figure 2: Exemplo de execução num ambiente multiprogramado

1.4 Modelo de Processos

Num ambiente **multiprogramado**, devido à constante **troca de processos**, é difícil expressar uma modelo para o processador. Devido ao elevado numero de processo e ao multiprogramming, torna-se difícil de saber qual o processo que está a ser executado e qual a fila de processos as ser executada.

É mais fácil assumir que o ambiente multiprogramado pode ser representado por um **conjunto de processadores virtuais**, estando um processo atribuído a cada um.

O processador virtual está: - **ON**: se o processo que lhe está atribuído está a ser executado - **OFF**: se o processo que lhe está atribuído não está a ser executado

Para este modelo temos ainda de assumir que: - Só um dos processadores é que pode estar ativo num dado período de tempo - O número de **processadores virtuais ativos é menor** (ou igual, se for um ambiente `single processor`) ao número de **processadores reais** - A execução de um processo **não é afetada** pelo instante temporal nem a localização no código em que o processo é interrompido e é efetuado o switching - Não existem restrições do número de vezes que qualquer processo pode ser interrompido, quer seja executado total ou parcialmente

A operação de **switching entre processos** e consequentemente entre processadores virtuais ocorre de forma não **controlada** pelo programa a correr no CPU

Uma **operação de switching** é equivalente a efetuar o `Turning Off` de um processo virtual e o `Turning On` de outro processo virtual.

- `Turning Off` implica **guardar** todo o **contexto de execução**
- `Turning On` implica carregar todo o contexto de execução, **restaurando o estado do programa** quando foi interrompido

1.5 Diagrama de Estados de um Processo

Durante a sua existência, um processo pode assumir diferentes estados, dependendo das condições em que se encontra:

- **run**: O processo está em execução, tendo a posse do processador
- **blocked**: O processo está bloqueado à **espera de um evento externo** para estar em condições retomar a sua execução. Esse evento externo pode ser:
 - Acesso a um recurso da máquina
 - Fim de uma operação de I/O
 - ...
- **ready**: O processo está pronto a ser executado, mas está à espera que o processador lhe dê a ordem de `start/resume` para poder retomar a sua execução.

As **transições entre estados** normalmente resultam de **intervenções externas ao processo**, mas podem depender de situações em que o processo força uma transição: - termina a sua execução antes de terminar o seu `time quantum` - Leitura/Escrita em I/O (`scanf/printf`)

Mesmo que um processo **não abandone o processador por sua iniciativa**, o `scheduler` é responsável por **planear o uso do processador pelos diferentes processos**.

O (`Process`) `Scheduler` é um módulo do kernel que **monitoriza e gere as transições entre processos**. Assim, um `while(1)` não é executado *ad eternum*. Um processador `multiprocess` só permite que o ciclo infinito seja executado quando é atribuído `CPU time` ao processo.

Existem diferentes políticas que permitem controlar a execução destas transições

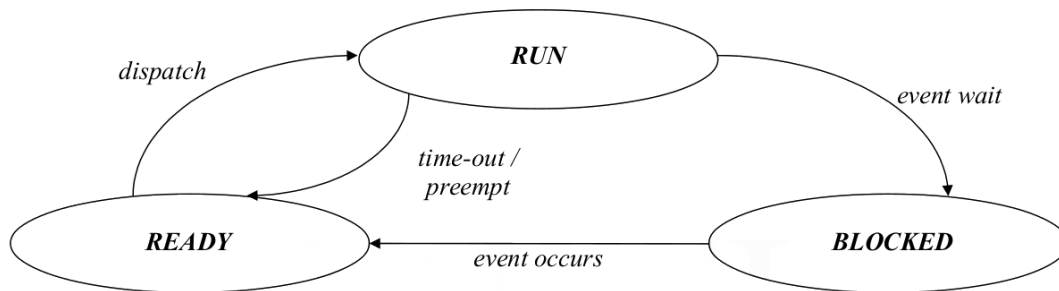


Figure 3: Diagrama de Estados do Processador - Básico

Triggers das transições entre estados:

- **dispatch:**

- O processo que estava em modo `run` perdeu o acesso ao processador.
- Do conjunto de processos prontos a serem executados, tem de ser escolhido **um** para ser executado, sendo lhe atribuído o processador.
- A escolha feita pelo `dispatcher` pode basear-se em:
 - * um sistema de prioridades
 - * requisitos temporais
 - * aleatoriedade
 - * divisão igual do CPU

- **event wait:**

- O processo que estava a ser executado sai do estado `run`, não estando em execução no processador.
 - * Ou porque é impedido de continuar pelo scheduler
 - * Ou por iniciativa do próprio processo.
 - `scanf`
 - `printf`
- O CPU guarda o estado de execução do processo
- O processo fica em estado `blocked` à **espera da ocorrência de um evento externo**, `event occurs`

- **event occurs:**

- Ocorreu o evento que o processo estava à espera
- O processo transita do estado `blocked` para o estado `ready`, ficando em fila de espera para que lhe seja atribuído o processador

- **time_out:**

- O processo esgotou a sua janela temporal, `time quantum`
- Através de uma interrupção em `hardware`, o sistema operativo vai forçar a saída do processo do processador

- Transita para o estado *ready* até lhe ser atribuído um novo *time-quantum* do CPU
- A transição por time out ocorre em qualquer momento do código.
- Os sistemas podem ter *time quantum* diferentes e os *time slots* alocados não têm de ser necessariamente iguais entre dois sistemas.

- **preempt:**

- O processo que possui a posse do processador tem uma prioridade mais baixa do que um processo que acordou e está pronto a correr (estado *ready*)
- O processo que está a correr no processador é **removido** e transita para o estado *ready*
- Passa a ser **executado** o processo de **maior prioridade**

1.5.1 Swap Area

O diagram de estados apresentado não leva em consideração que a **memória principal** (RAM) é **finita**. Isto implica que o número de **processos coexistentes em memória é limitado**.

É necessário usar a **memória secundária** (Disco Rígido) para **extender a memória principal** e aumentar a capacidade de armazenamento dos estados dos processos.

A **memória swap** pode ser:

- uma partição de um disco
- um ficheiro

Qualquer processo que **não esteja a correr** por ser *swapped out*, libertando memória principal para outros processos

Qualquer processo *swapped out* pode ser *swapped in*, **quando existir memória principal disponível**

Ao diagrama de estados tem de ser adicionados: - dois novos estados: - **suspended-ready**: Um processo no estado *ready* foi *swapped-out* - **suspended-blocked**: O processo no estado *blocked* foi *swapped-out* - dois novos tipos de transições: - **suspend**: O processo é *swapped out* - **activate**: O processo é *swapped in*

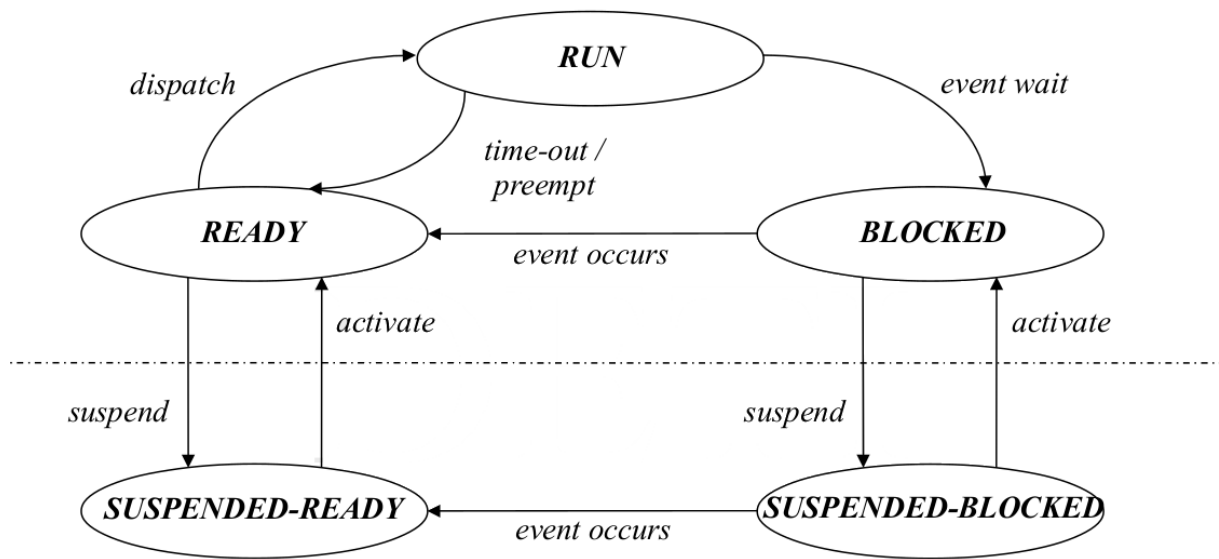


Figure 4: Diagrama de Estados do Processador - Com Memória de Swap

1.5.2 Temporalidade na vida dos processos

O diagrama assume que os processos são **intemporais**. Excluindo alguns processos de sistema, todos os processos são **temporais**, i.e.:

1. Nascem/São criados
2. Existem (por algum tempo)
3. Morrem/Terminam

Para introduzi a temporalidade no diagrama de estados, são necessários dois novos estados: - **new**: - O processo foi criado - Ainda não foi atribuído à **pool** de processos a serem executados - A estrutura de dados associado ao processo é inicializada - **terminated**: - O processo foi descartado da fila de processos executáveis - Antes de ser descartado, existem ações que tem de tomar (*needs clarification*)

Em consequência dos novos estados, passam a existir três novas transições: - **admit**: O processo é admitido pelo OS para a **pool** de processos executáveis - **exit**: O processo informa o SO que terminou a sua execução - **abort**: Um processo é forçado a terminar. - Ocorreu um **fatal error** - Um processo autorizado abortou a sua execução

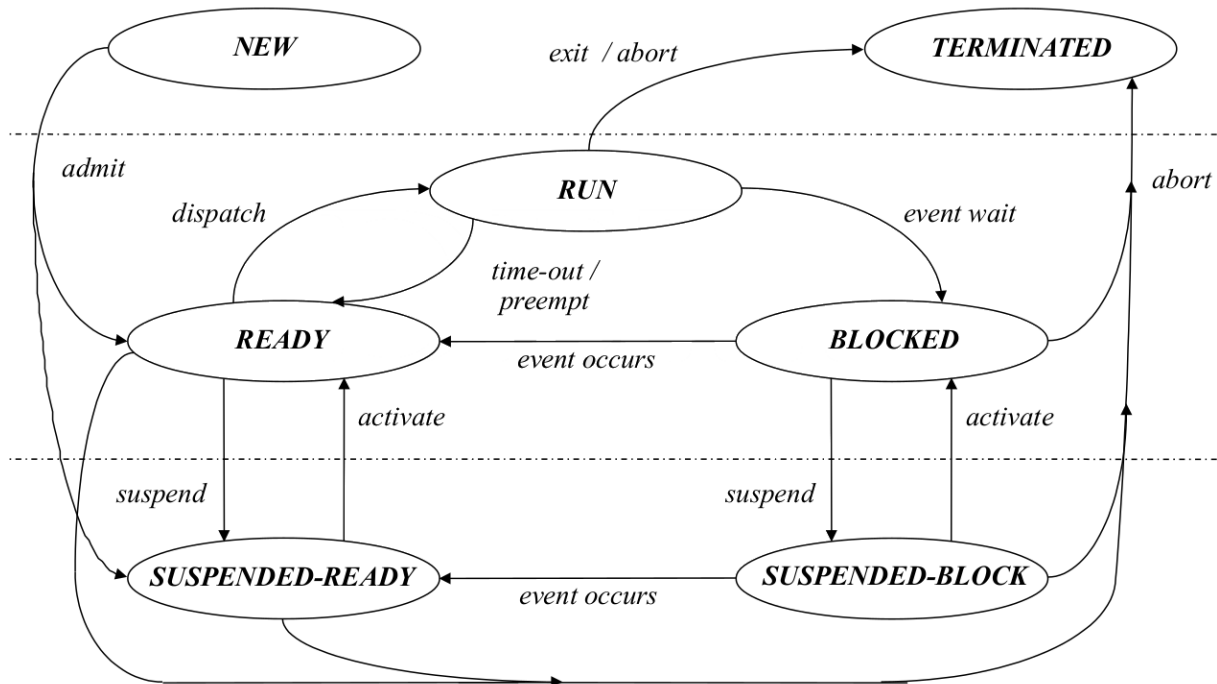


Figure 5: Diagrama de Estados do Processador - Com Processos Temporalmente Finitos

1.6 State Diagram of a Unix Process

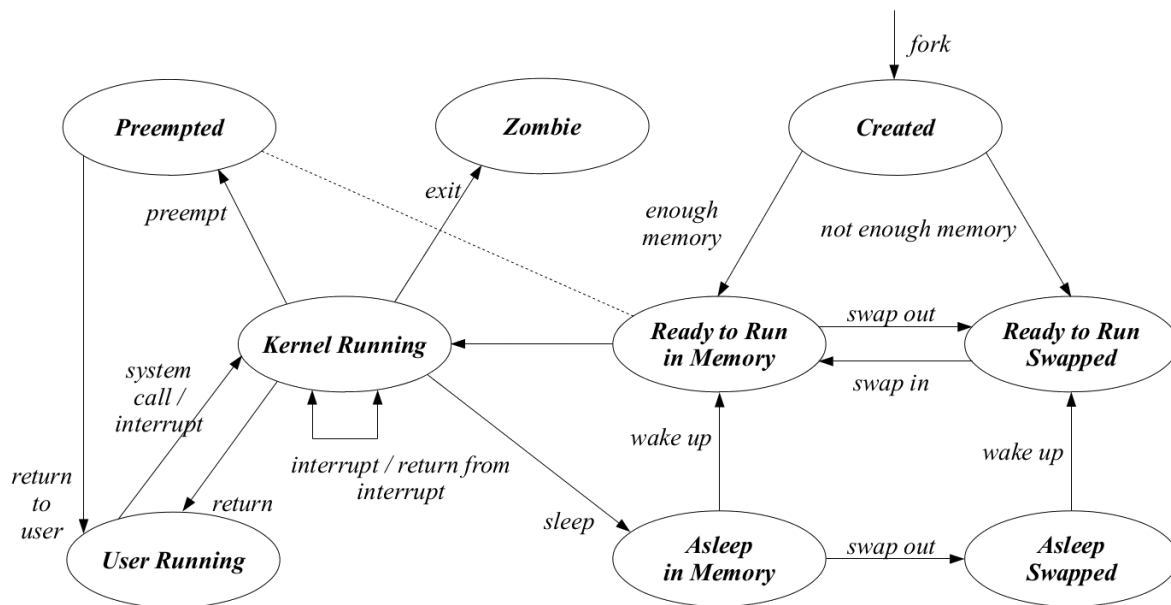


Figure 6: Diagrama de Estados do Processador - Com Memória de Swap

As três diferenças entre o diagrama de estados de um processo e o diagrama de estados do sistema Unix são

1. Existem **dois estados run**

1. **kernel running**
2. **user running**

- Diferem **no modo** como o processador **executa o código máquina**, existindo **mais instruções e diretivas disponíveis** no modo supervisor (*root*)

2. O estado **ready** é dividido em dois estados:

1. **ready to run in memory:** O processo está pronto para ser executado/continuar a execução, estando guardado o seu estado em memória
2. **preempted:** O processo que estava a ser executado foi **forçado a sair do processador** porque **existe um processo mais prioritário para ser executado**

- Estes **estados são equivalentes** porque:

- estão ambos **armazenado na memória principal**
- quando um processo é *preempted* continua pronto a ser executado (não precisando de nenhuma informação de I/O)
- Partilham a mesma fila (*queue*) de processos, logo são tratados de forma idêntica pelo OS
- Quando um **processo do utilizador abandona o modo de supervisor** (corre com permissões *root*), **pode ser preempted**

3. A transição de `time-out` que existe no diagrama dos estados de um processo em UNIX é coberta pela transição `preempted`

1.7 Supervisor preempting

Tradicionalmente, a **execução** de um processo **em modo supervisor** (`root`) implicava que a execução do processo **não pudesse ser** interrompida, ou seja, o processo não pode ser `preempted`. Ou seja, o UNIX não permitia **real-time processing**

Nas novas versões o código está dividido em **regiões atômicas**, onde a **execução não pode ser interrompida** para garantir a **preservação de informação das estruturas de dados a manipular**. Fora das regiões atômicas é seguro interromper a execução do código

Cria-se assim uma nova transição, `return to kernel` entre os estados `preempted` e `kernel running`.

1.8 Unix – traditional login

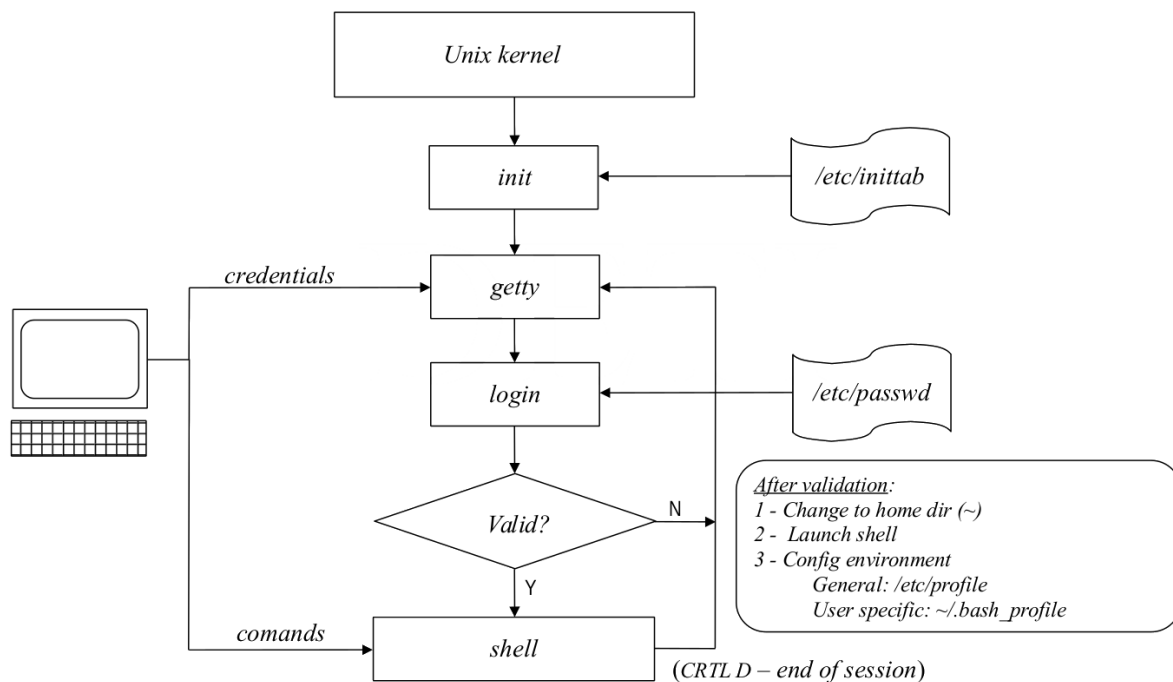


Figure 7: Diagrama do Login em Linux

1.9 Criação de Processos

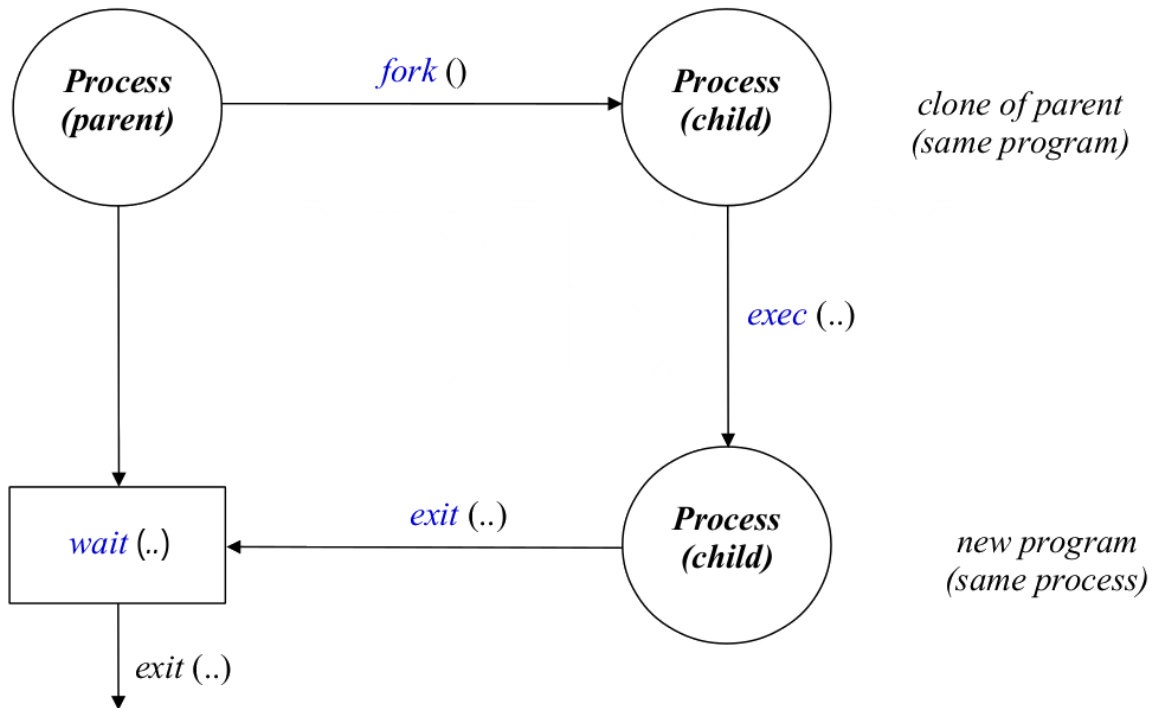


Figure 8: Criação de Processos

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     printf("Before the fork:\n");
9     printf(" PID = %d, PPID = %d.\n",
10         getpid(), getppid());
11
12     fork();
13
14     printf("After the fork:\n");
15     printf(" PID = %d, PPID = %d. Who am I?\n", getpid(), getppid());
16
17     return EXIT_SUCCESS;
18 }
  
```

- **fork: clona** o processo existente, criando uma **réplica**
 - O estado de execução é igual, incluindo o PC (*Program Counter*)
 - O **mesmo programa** é executado em **processos diferentes**
 - Não existem garantias que o pai execute primeiro que o filho
 - * Tudo depende do `time quantum` que o processo pai ocupa antes do `fork`
 - * É um ambiente multiprogramado: os dois programas correm em **simultâneo no micro tempo**
- O **espaço de endereçamento** dos dois processos é **igual**
 - É seguida uma filosofia **copy-on-write**. Só é efetuada a cópia da página de memória se o **processo filho modificar** os conteúdos das variáveis

Existem variáveis diferentes:

- **PPID:** Parent Process ID
- **PID:** Process ID

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     printf("Before the fork:\n");
9     printf(" PID = %d, PPID = %d.\n",
10    getpid(), getppid());
11
12    int ret = fork();
13
14    printf("After the fork:\n");
15    printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);
16
17    return EXIT_SUCCESS;
18 }
```

O retorno da instrução `fork` é diferente entre o processo pai e filho:

- pai: PID do filho
- filho: 0

O retorno do `fork` pode ser usado como variável booleana, de modo a **distinguir o código a correr no filho e no pai**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
```

```
6 int main(void)
7 {
8     printf("Before the fork:\n");
9     printf(" PID = %d, PPID = %d.\n", getpid(), getppid());
10
11     int ret = fork();
12
13     if (ret == 0)
14     {
15         printf("I'm the child:\n");
16         printf(" PID = %d, PPID = %d\n", getpid(), getppid());
17     }
18     else
19     {
20         printf("I'm the parent:\n");
21         printf(" PID = %d, PPID = %d\n", getpid(), getppid());
22     }
23
24     printf("After the fork:\n");
25     printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);
26
27     return EXIT_SUCCESS;
28 }
```

O `fork` por si só não possui grande interesse. O interesse é poder executar um programa diferente no filho.

- **exec system call:** Executar um programa diferente no processo filho
- **wait system call:** O pai esperar pela conclusão do programa a correr nos filhos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     /* check arguments */
9     if (argc != 2)
10    {
11        fprintf(stderr, "spawn <path to file>\n");
12        exit(EXIT_FAILURE);
13    }
14    char *aplic = argv[1];
15
16    printf("=====\n");
17
18    /* clone phase */
19    int pid;
```

```
20     if ((pid = fork()) < 0)
21     {
22         perror("Fail cloning process");
23         exit(EXIT_FAILURE);
24     }
25
26     /* exec and wait phases */
27     if (pid != 0) // only runs in parent process
28     {
29         int status;
30         while (wait(&status) == -1);
31         printf("=====\n");
32         printf("Process %d (child of %d)"
33             "ends with status %d\n",
34             pid, getpid(), WEXITSTATUS(status));
35     }
36     else // this only runs in the child process
37     {
38         execl(aplic, aplic, NULL);
39         perror("Fail launching program");
40         exit(EXIT_FAILURE);
41     }
42 }
```

O fork pode **não ser bem sucedido**, ocorrendo um `fork failure`.

1.10 Execução de um programa em C/C++

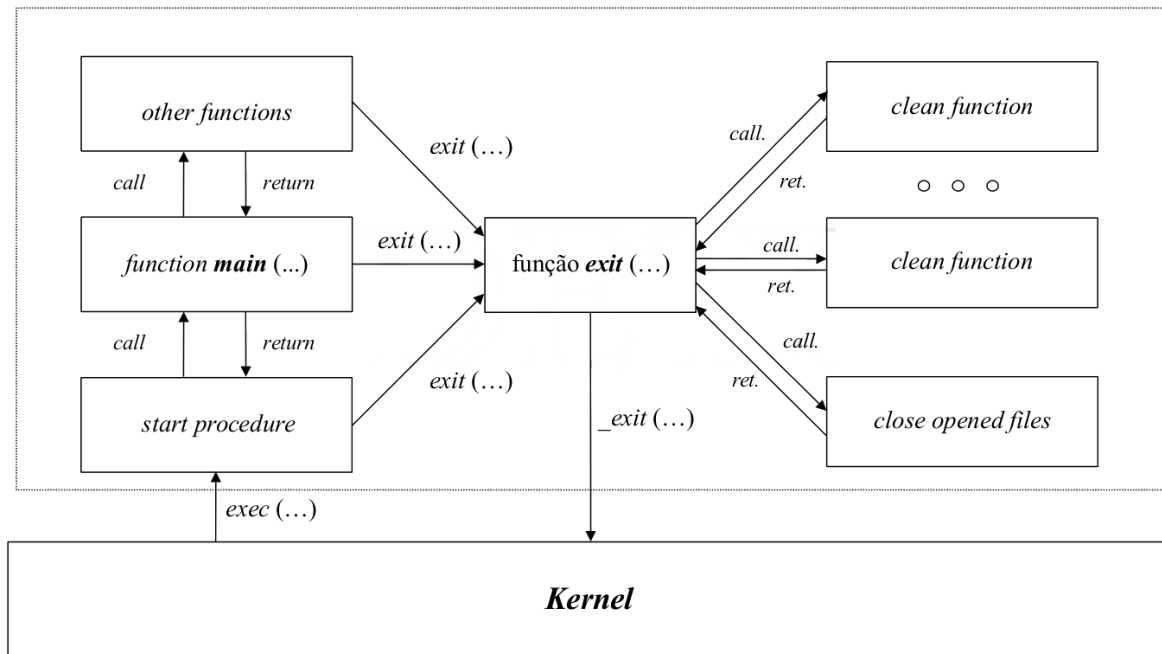


Figure 9: Execução de um programa em C/C++

- Em C/C++ o nome de uma função é um ponteiro para a sua função.
- Em C/C++ um include não inclui a biblioteca
 - Indica ao programa que vou usar uma função que tem esta assinatura
- **atexit:** Regista uma função para ser chamada no fim da execução normal do programa
 - São chamadas em ordem inversa ao seu registo

1.11 Argumentos passados pela linha de comandos e variáveis de ambiente

- **argv:** array de strings que representa um conjunto de parâmetros passados ao programa
 - **argv[0]** é a referência do programa
- **env** é um array de strings onde cada string representa uma variável do tipo: `name=value`
- **getenv** devolve o valor de uma variável definida no array **env**

1.12 Espaço de Endereçamento de um Processo em Linux

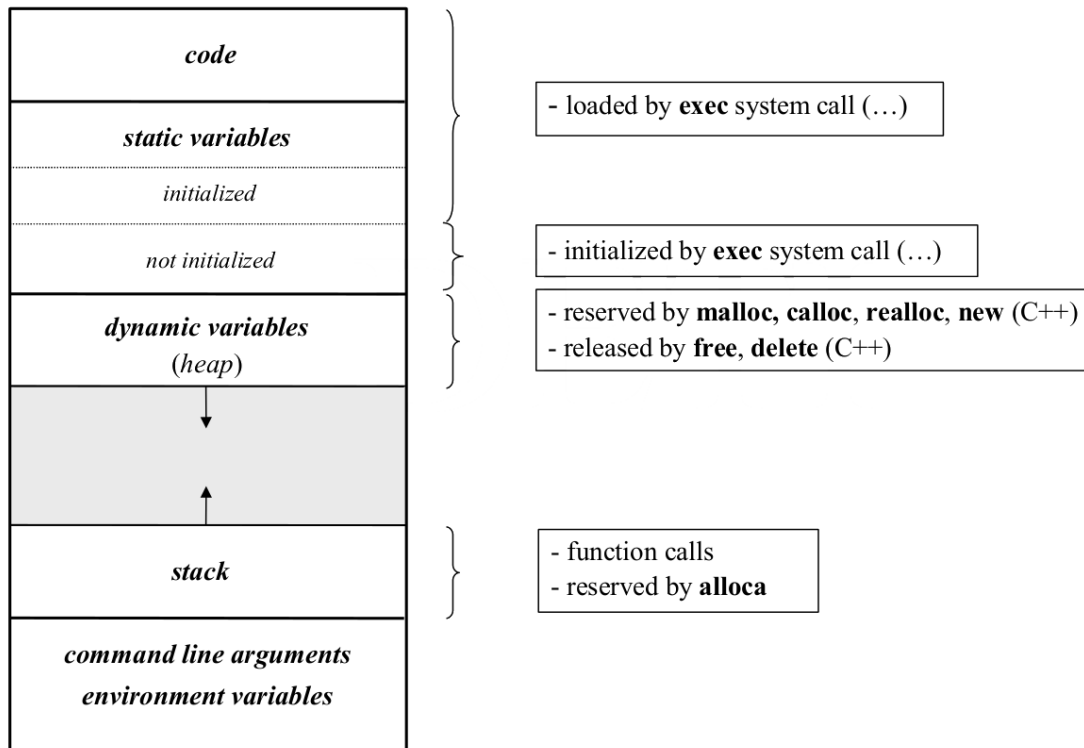


Figure 10: Espaço de endereçamento de um processo em Linux

- As variáveis que existem no processo pai também existem no processo filho (clonadas)
- As variáveis globais são comuns aos dois processos
- Os endereços das variáveis são todos iguais porque o espaço de endereçamento é igual (memória virtual)
- Cada processo tem as suas variáveis, residindo numa página de memória física diferente
- Quando o processo é clonado, o espaço de dados só é clonado quando um processo escreve numa variável, ou seja, após a modificação é que são efetuadas as cópias dos dados
- O programa acede a um endereço de memória virtual e depois existe hardware que trata de alocar esse endereço de memória de virtual num endereço de memória física
- Posso ter dois processos com memórias virtuais distintas mas fisicamente estarem ligados *ao mesmo endereço de memória*
- Quando faço um `fork` não posso assumir que existem variáveis partilhadas entre os processos

1.12.1 Process Control Table

É um array de `process control block`, uma estrutura de dados mantida pelo sistema operativo para guardar a informação relativa todos os processos.

O `process controlo block` é usado para guardar a informação relativa a apenas um processo, possuindo os campos:

- `identification`: id do processo, processo-pai, utilizador e grupo de utilizadores a que pertence
- `address space`: endereço de memória/swap onde se encontra:
 - código
 - dados
 - stack
- `processo state`: valor dos registos internos (incluindo o PC e o `stack pointer`) quando ocorre o switching entre processos
- `I/O context`: canais de comunicação e respetivos buffers que o processo tem associados a si
- `state`: estado de execução, prioridade e eventos
- `statistic`: *start time, CPU time*

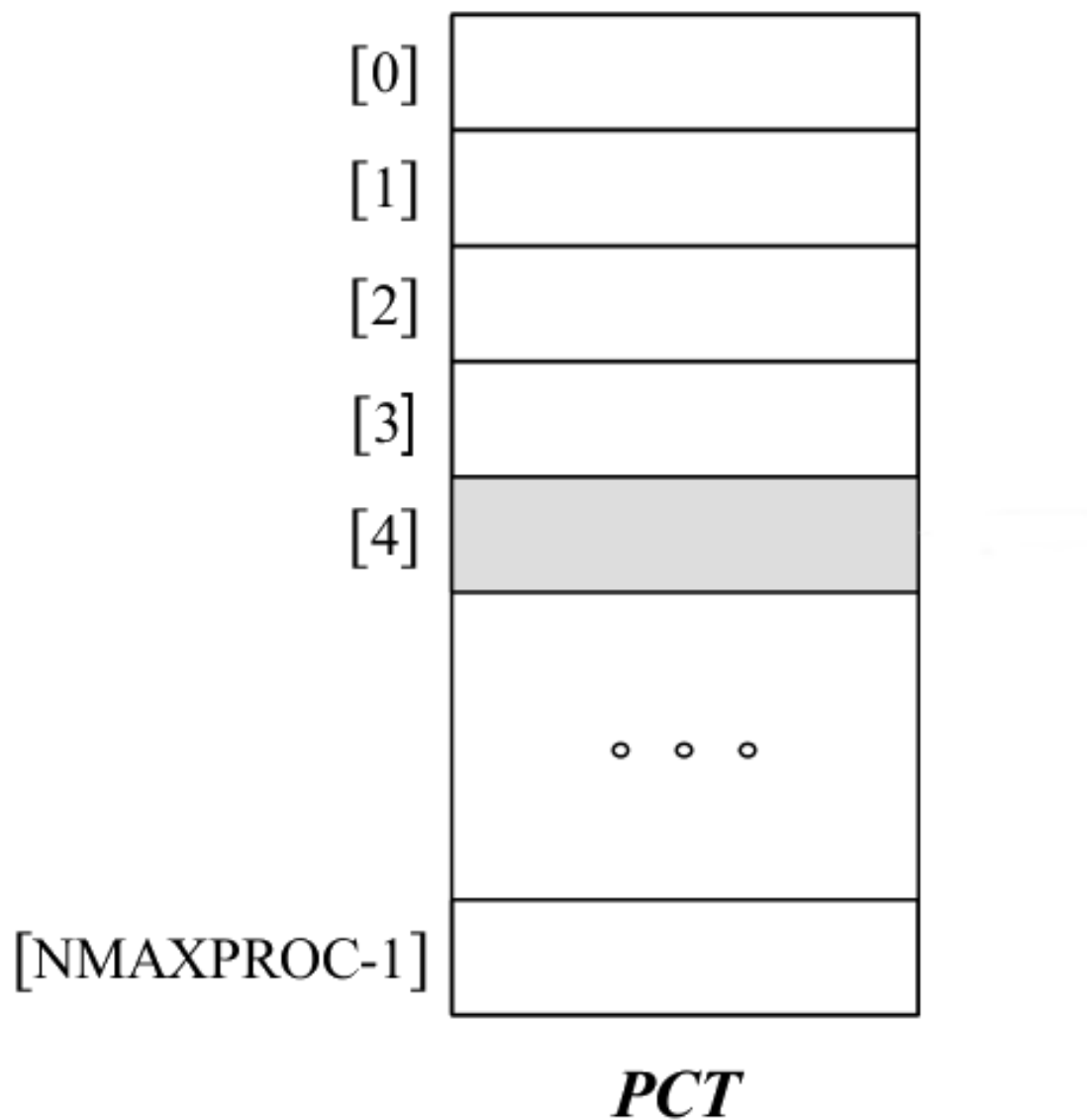


Figure 11: Process Control Table

2 Threads

Num sistema operativo tradicional, um **processo** inclui:

- um **espaço de endereçamento**
- um **conjunto de canais de comunicação** com dispositivos de I/O

- uma única **thread de controlo** que:
 - incorpora os **registos do processador** (incluindo o PC)
 - **stack**

Existem duas stacks no sistema operativo:

- **user stack**: cada **processo/thread** possui a sua (em memória virtual e corre com privilégios do **user**)
- **system stack**: global a todo o sistema operativo (no **kernel**)

Podendo estes dois componentes serem **geridos de forma independente**.

Visto que uma **thread** é apenas um **componente de execução** dentro de um processo, várias **threads independentes** podem coexistir no mesmo processo, **partilhando** o mesmo **espaço de endereçamento** e o mesmo contexto de **acesso aos dispositivos de I/O**. Isto é **multithreading**.

Na prática, as **threads** podem ser vistas como *light weight processes*

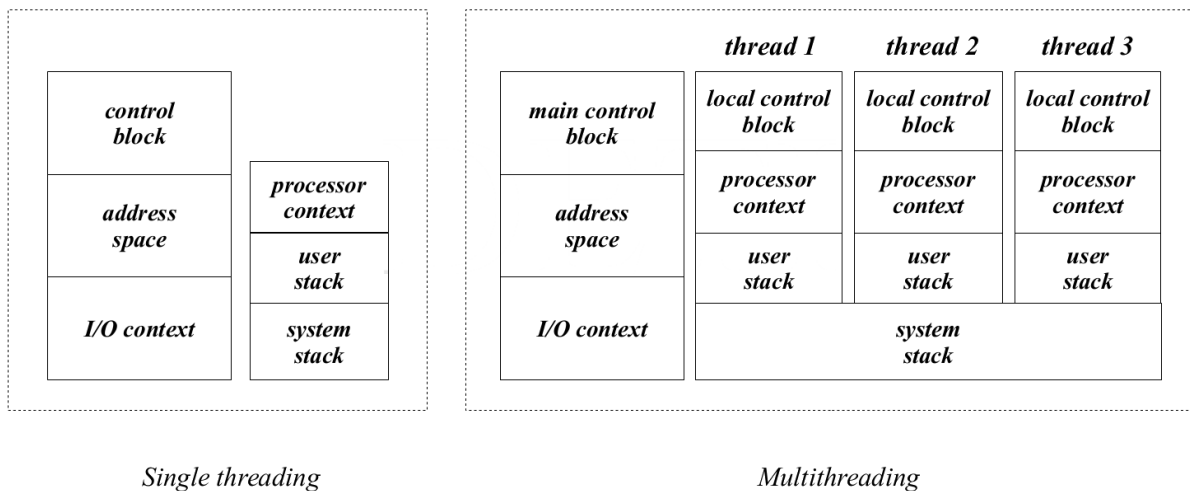


Figure 12: Single threading vs Multithreading

O controlo passa a ser centralizado na **thread** principal que gere o processo. A **user stack**, o **contexto de execução do processador** passa a ser dividido por todas as **threads**.

2.1 Diagrama de Estados de uma thread

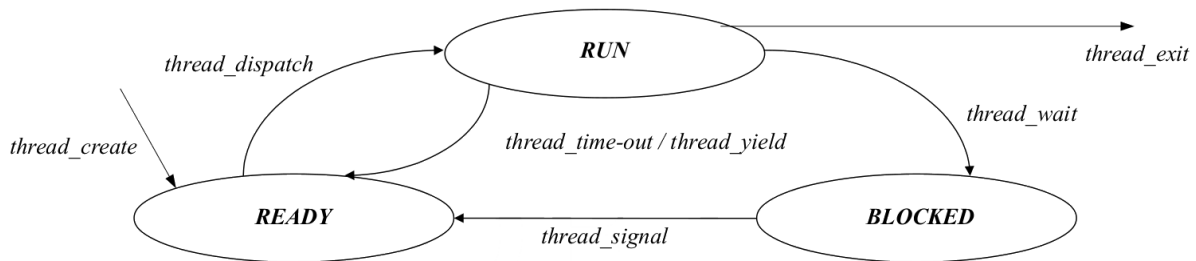


Figure 13: Diagrama de estados de uma thread

O diagrama de estados de um `thread` é mais simplificado do que o de um processo, porque só são “necessários” os estados que interagem **diretamente com o processador**:

- 1 - ‘run‘
- 2 - ‘ready‘
- 3 - ‘blocked‘

Os estados `suspend-ready` e `suspended-blocked` estão relacionados com o **espaço de endereçamento do processo** e com a zona onde estes dados estão guardados, dizendo respeito ao **processo e não à thread**

Os estados `new` e `terminated` não estão presentes, porque a gestão do ambiente multiprogramado prende-se com a restrição do número de `threads` que um processo pode ter, logo dizem respeito ao processo

2.2 Vantagens de Multithreading

- **facilita a implementação** (em certas aplicações):
 - Existem aplicações em que **decompor a solução** num conjunto de `threads` que correm paralelamente facilita a implementação
 - Como o `address space` e o `I/O context` são partilhados por todas as `threads`, `multithreading` favorece esta decomposição
- **melhor utilização dos recursos**
 - A criação, destruição e `switching` é mais eficiente usando `threads` em vez de processos
- **melhor performance**
 - Em aplicações `I/O driven`, `multithreading` permite que **várias atividades se sobreponham, aumentando a rapidez** da sua execução
- **multiprocessing**
 - É possível **paralelismo em tempo real** se o processador possuir **múltiplos CPUs**

2.3 Estrutura de um programa multithreaded

- Cada `thread` está tipicamente associada com a execução de uma função que implementa alguma atividade em específico
- A **comunicação entre threads** é efetuada através da estrutura de dados do **processo**, que é vista pelas `threads` como uma estrutura de dados global
- o **programa principal** também é uma `thread`
 - A 1ª a ser criada
 - Por norma a última a ser destruída

2.4 Implementação de Multithreading

user level threads:

- Implementadas por uma biblioteca
 - Suporta a criação e gestão das `threads` sem intervenção do `kernel`
- Correm com permissões do utilizador
- Solução versátil e portátil
- Quando uma `thread` executa uma `system call` bloqueante, **todo o processo bloqueia** (o `kernel` só “vê” o processo)
- Quando passo variáveis a `threads`, elas têm de ser estáticas ou dinâmicas

kernel level threads

- As `threads` são implementadas diretamente ao nível do `kernel`
- Menos versáteis e portáteis
- Quando uma `thread` executa uma `system call` bloqueante, **outra thread pode entrar em execução**

2.4.1 Libreria pthread

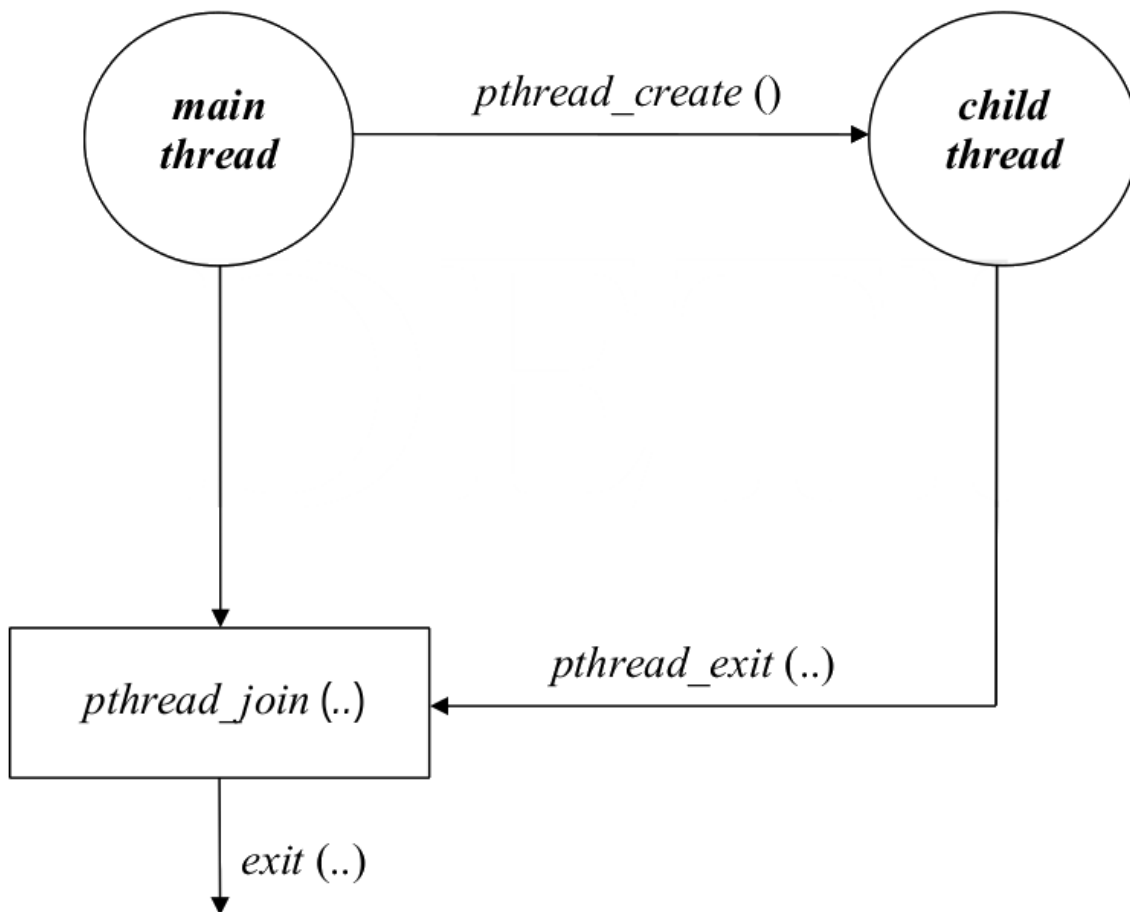


Figure 14: Exemplo do uso da biblioteca pthread

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 /* return status */
6 int status;
7
8 /* child thread */
9 void *threadChild (void *par)
10 {
11     printf ("I'm the child thread!\n");
12     status = EXIT_SUCCESS;
13     pthread_exit (&status);
14 }
```



```
15
16 /* main thread */
17 int main (int argc, char *argv[])
18 {
19     /* launching the child thread */
20     pthread_t thr;
21     if (pthread_create (&thr, NULL, threadChild, NULL) != 0)
22     {
23         perror ("Fail launching thread");
24         return EXIT_FAILURE;
25     }
26
27     /* waits for child termination */
28     if (pthread_join (thr, NULL) != 0)
29     {
30         perror ("Fail joining child");
31         return EXIT_FAILURE;
32     }
33
34     printf ("Child ends; status %d.\n", status);
35     return EXIT_SUCCESS;
36 }
```

2.5 Threads em Linux

2 `system calls` para criar processos filhos:

- `fork`:
 - cria um novo processo que é uma **cópia integral** do processo atual
 - o `address space` e `I/O context` é duplicado
- `clone`:
 - cria um novo processo que pode partilhar elementos com o pai
 - Podem ser partilhados
 - * espaço de endereçamento
 - * tabela de `file descriptors`
 - * tabela de `signal handlers`
 - O processo filho executa uma dada função

Do ponto de vista do `kernel`, `processos` e `threads` são **tratados de forma semelhante**

`Threads` do **mesmo processo** forma um `thread group` e possuem o **mesmo thread group identifier** (TGID). Este é o valor retornado pela função `getpid()` quando aplicada a um grupo de `threads`

As várias `threads` podem ser distinguidas dentro de um **grupo de threads** pelo seu `unique thread identifier` (TID). É o valor retornado pela função `gettid()`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7
8 pid_t gettid()
9 {
10     return syscall(SYS_gettid);
11 }
12
13 /* child thread */
14 int status;
15 void *threadChild (void *par)
16 {
17     /* There is no glibc wrapper, so it was to be called
18      * indirectly through a system call
19      */
20
21     printf ("Child: PPID: %d, PID: %d, TID: %d\n", getppid(), getpid(), gettid
22            ());
23     status = EXIT_SUCCESS;
24     pthread_exit (&status);
25 }
```

O TID da *main thread* é a mesma que o PID do processo, **porque são a mesma entidade**.

Para efetuar a compilação, tenho de indicar que a biblioteca pthread tem de ser usada na linkagem:

```
1 g++ -o x thread.cpp -pthread
```

3 Process Switching

Revisitando a o diagrama de estados de um processador [multithreading](#)

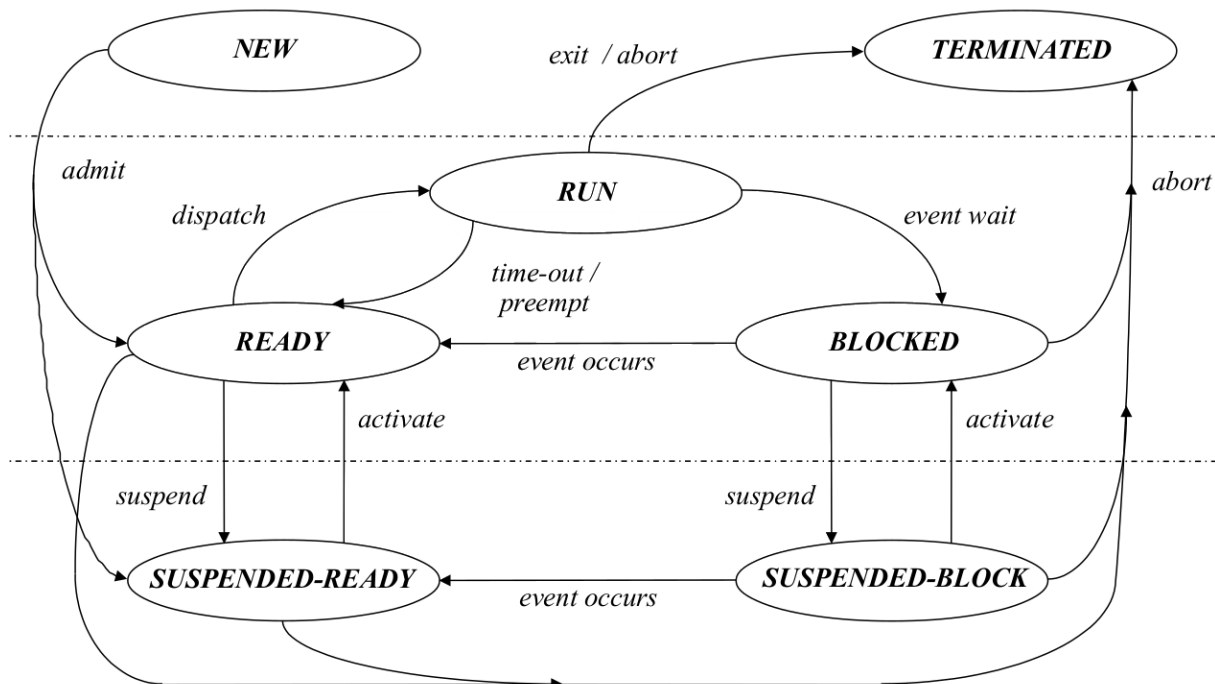


Figure 15: Diagrama de estados completo para um processador multithreading

Os processadores atuais possuem **dois modos de funcionamento**:

1. *supervisor mode*

- Todas as instruções podem ser executadas
- É um modo **privilegiado, reservado para o sistema operativo**
- O modo em que o **sistema operativo devia funcionar**, para garantir que pode aceder a todas as funcionalidades do processador

2. *user mode*

- Só uma **secção reduzida do instruction set** é que pode ser executada
- Instruções de I/O e outras que modifiquem os registos de controlo não são executadas em *user mode*
- É o **modo normal de operação**

A **troca entre os dois modos de operação**, *switching*, só é possível através de *exceções*. Uma *exceção* pode ser causada por:

- Interrupção de um dispositivo de I/O
- Instrução ilegal
 - divisão por zero
 - bus error
 - ...

- trap instruction (aka interrupção por *software*)

As **funções do kernel**, incluindo as *system calls* só podem ser lançadas por:

- **hardware** \Rightarrow interrupção
- **traps** \Rightarrow interrupção por *software*

O ambiente de operação nestas condições é denominado de *exception handling*

3.1 Exception Handling

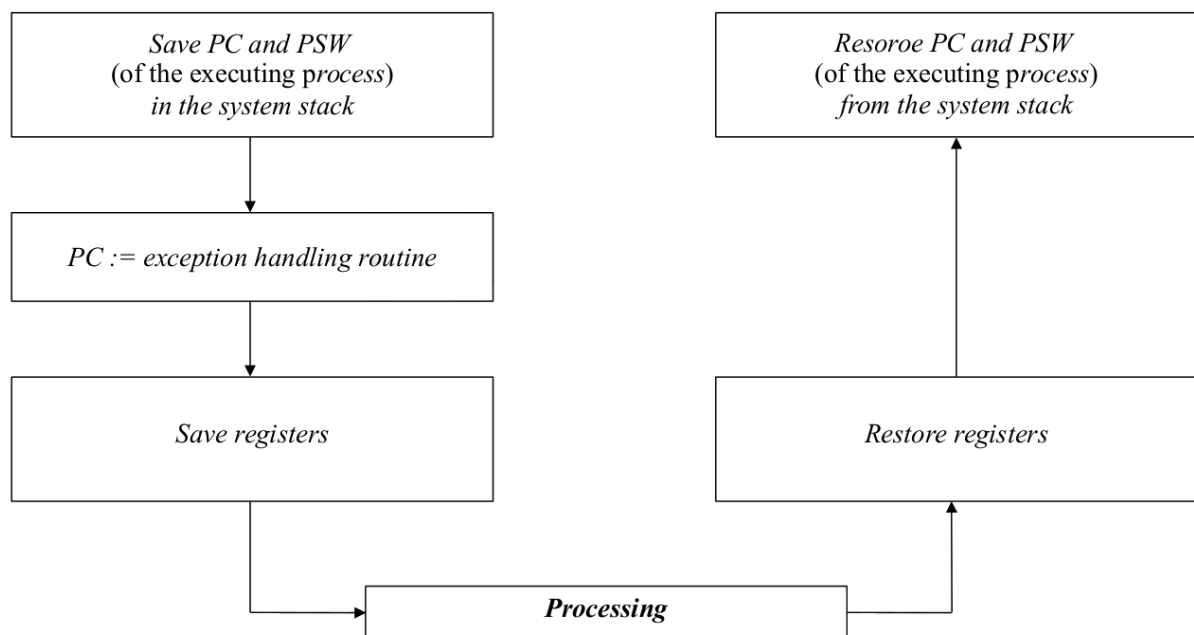


Figure 16: Algoritmo a seguir para tratar de exceções normais

A **troca do contexto de execução** é feita guardando o estado dos registos PC e PSW na stack do sistema, saltando para a rotina de interrupção e em seguida salvaguardando os registos que a rotina de tratamento da exceção vai precisar de modificar. No fim, os valores dos registos são restaurados e o programa resume a sua execução

3.2 Processing a process switching

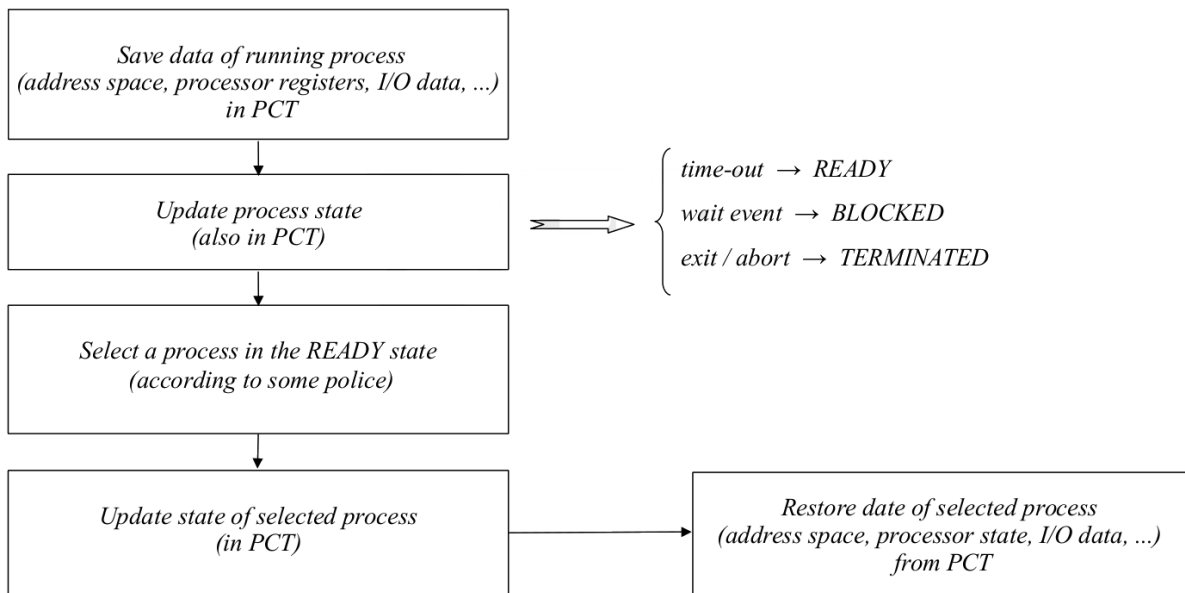


Figure 17: Algoritmo a seguir para efetuar uma process switching

O algoritmo é bastante parecido com o tratamento de exceções:

1. Salvar todos os dados relacionados com o processo atual
2. Efetuar a troca para um novo processo
3. Correr esse novo processo
4. Restaurar os dados e a execução do processo anterior

4 Processor Scheduling

A execução de um processo é uma sequência alternada de períodos de:

- **CPU burst**, causado pela execução de instruções do CPU
- **I/O burst**, causados pela espera do resultado de pedidos a dispositivos de I/O

O processo pode então ser classificado como:

- **I/O bound** se possuir muitos e curtos **CPU bursts**
- **CPU bound** se possuir poucos e longos **CPU bursts**

O objetivo da **multiprogramação** é obter vantagem dos períodos de **I/O burst** para permitir outros processos terem acesso ao processador. A componente do sistema responsável por esta gestão é o **scheduler**.

A funcionalidade principal do **scheduler** é decidir da **poll** de processos prontos para serem executados que coexistem no sistema:

- quais é que devem ser executados?
- quando?
- por quanto tempo?
- porque ordem?

4.1 Scheduler

Revisitando o diagrama de estados do processador, identificamos três schedulers

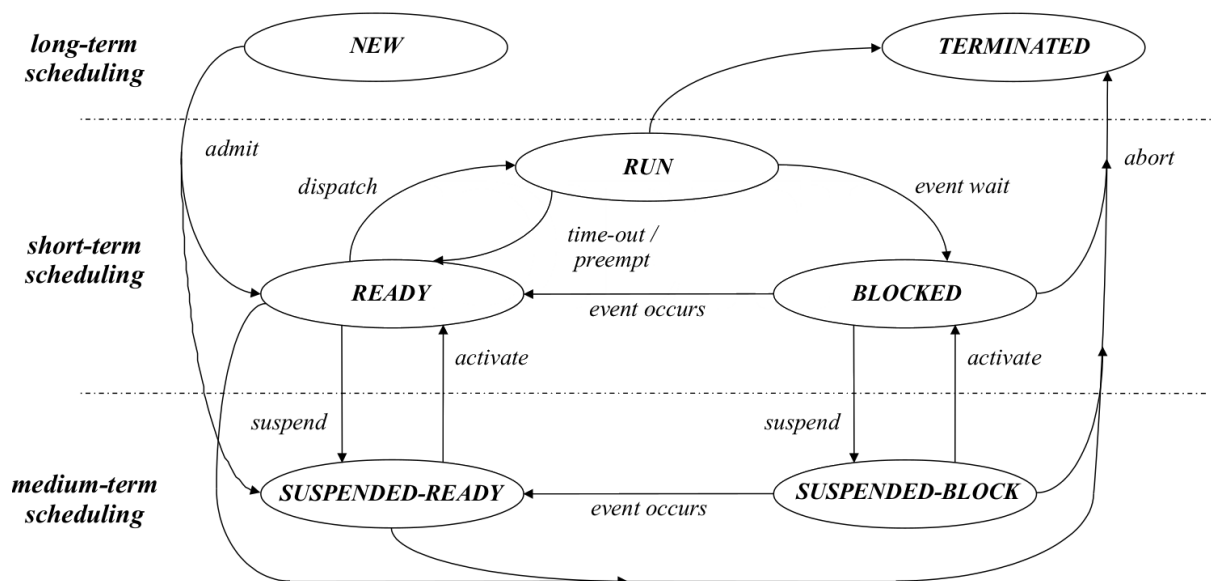


Figure 18: Identificação dos diferentes tipos de schedulers no diagrama de estados dos processos

4.1.1 Long-Term Scheduling

Determina que **programas são admitidos para serem processados:**

- Controla o **grau de multiprogramação** do sistema
- Se um programa do utilizador ou **job** for aceite, torna-se um **processo** e é adicionado à **queue de processos ready em fila de espera**
 - Em princípio é adicionado à **queue do short-term scheduler**
 - mas também é possível que seja adicionada à **queue do medium-term scheduler**
- Pode colocar processos em **suspended ready**, libertando quer a memória quer a fila de processos

4.1.2 Medium Term Scheduling

Gere a **swapping area**

- As decisões de `swap-in` são **controladas pelo grau de multiprogramação**
- As decisões de `swap-in` são **condicionadas pela gestão de memória**

4.1.3 Short-Term Scheduling

Decide qual o **próximo processo a executar**

- É invocado quando existe um evento que:
 - **bloqueia o processo atual**
 - **permite que este seja preempted**
- Eventos possíveis são:
 - interrupção de relógio
 - interrupção de I/O
 - `system calls`
 - signal (e.g. através de semáforos)

4.2 Critérios de Scheduling

4.2.1 User oriented

Turnaround Time:

- Intervalo de de tempo entre a submissão de um processo até à sua conclusão
- Inclui:
 - Tempo de execução enquanto o processo tem a posse do CPU
 - Tempo dispendido à espera pelos recursos que precisa (inclui o processador)
- Deve ser minimizado em sistemas `batch`
- É a medida apropriada para um `batch job`

Waiting Time:

- Soma de todos os períodos de tempo em que o processo esteve à espera de ser colocado no estado `ready`
- Deve ser minimizado

Response Time:

- Intervalo de tempo que decorre desde a submissão de um pedido até a resposta começa a ser produzida
- Medida apropriada para sistemas/processos interativo
- Deve ser minimizada para este tipo de sistemas/processos
- O número de processos interativos deve ser maximizado desde que seja garantido um tempo de resposta aceitável

Deadlines:

- Tempo necessário para um processo terminar a sua execução

- Usado em sistemas de tempo real
- A percentagem de **deadlines** atingidas deve ser maximizada, mesmo que isso implique subordinar/reduzir a importância de outros objetivos/parâmetros do sistema

Predictability:

- Quantiza o impacto da carga (de processos) no tempo de resposta dos sistema
- Idealmente, um **job** deve correr no **mesmo intervalo de tempo** e gastar os **mesmos recursos de sistema** independentemente da carga que o sistema possui

4.2.2 System oriented**Fairness:**

- Igualdade de tratamento entre todos os processos
- Na ausência de diretivas que condicionem os processos a atender, deve ser efetuada um gestão e partilha justa dos recursos, onde todos os processos são tratados de forma equitativa
- Nenhum processo pode sofrer de **starvation**

Throughput:

- Medida do número de processos completados por unidade de tempo (“taxa de transferência” de processos)
- Mede a quantidade de trabalho a ser executada pelos processos
- Deve ser maximizado
- Depende do tamanho dos processos e da **política de escalonamento**

Processor Utilization:

- Mede a percentagem que o processador está ocupado
- Deve ser maximizada, especialmente em sistemas onde predomina a partilha do processador

Enforcing Priorities:

- Os processos de **maior prioridade** devem ser sempre favorecidos em detrimento de processos menos prioritários

É impossível favorecer todos os critérios em simultâneo

Os **critérios a favorecer** dependem da **aplicação específica**

4.3 Preemption & Non-Preemption**Non-preemptive scheduling:**

- O processo mantém o processador até este ser bloqueado ou terminar
- As **transições são sempre por time-out**
- Não existe **preempt**

- Típico de sistemas *batch*
 - Não existem deadlines nem limitações temporais restritas a cumprir

Preemptive Scheduling:

- O processo pode **perder o processador devido a eventos externos**
 - esgotou o seu *time-quantum*
 - um processo **mais prioritário** está *ready*
 - Típico de **sistemas interativos**
 - * É preciso garantir que a resposta ocorre em intervalos de tempo limitados
 - * É preciso “simular” a ideia de paralelismo no *macro-tempo*
 - Sistemas em tempo real são *preemptive* porque existem *deadlines restritas* que precisam de ser cumpridas
 - Nestas situações é importante que um **evento externo** tenha capacidade de libertar o processador

4.4 Scheduling

4.4.1 Favouring Fairness

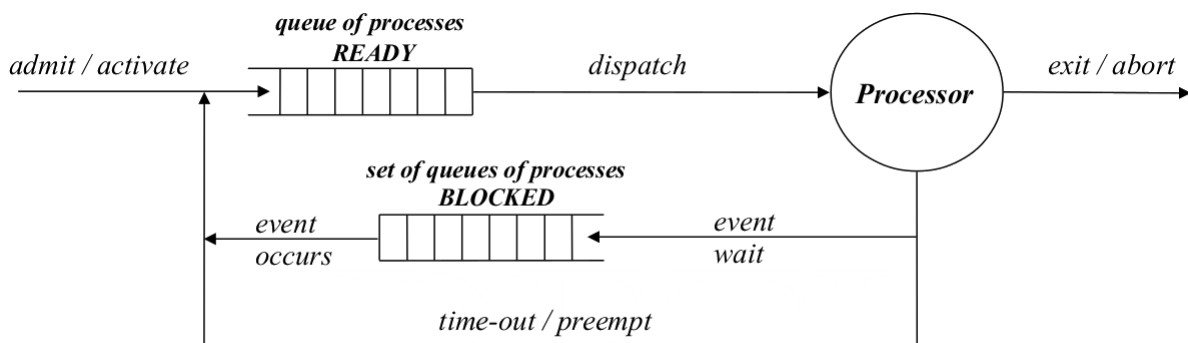


Figure 19: Espaço de endereçamento de um processo em Linux

Todos os processos **são iguais** e são atendidos por **ordem de chegada**

- É implementado usando **FIFOs**
- Pode existir mais do que um processo à espera de eventos externos
- Existe uma fila de espera para cada evento
- Fácil de implementar
- Favorece processos *CPU-bound* em detrimento de processos *I/O-bound*
 - Só necessitam de acesso ao processador, não de recursos externos
 - Se for a vez de um processo *I/O-bound* ser atendido e não possuir os recursos de I/O que precisa tem de voltar para a fila

- Em **sistemas interativos**, o `time-quantum` deve ser escolhido cuidadosamente para obter um bom compromisso entre `fairness` e `response time`

Em função do scheduling pode ser definido como:

- `non-preemptive scheduling` \Rightarrow `first come, first-served` (FCFS)
- `preemptive scheduling` \Rightarrow `round robin`

4.4.2 Priorities

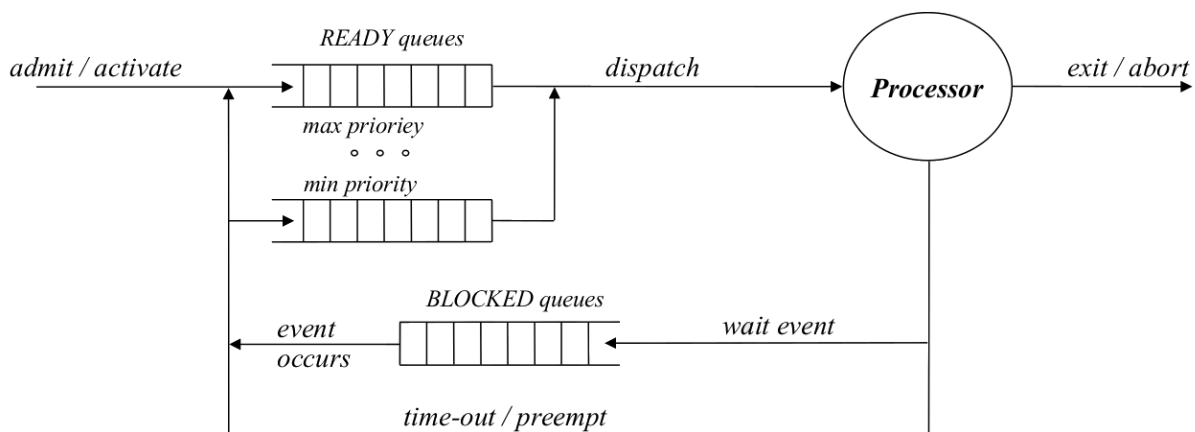


Figure 20: Espaço de endereçamento de um processo em Linux

Segue o princípio de que atribuir a mesma importância a todos os processos pode ser uma solução errada. Um sistema injusto *per se* não é necessariamente mau.

- A **minimização do tempo de resposta** (`response time`) exige que os processos `I/O-bound` sejam **privilegiados**
- Em **sistemas de tempo real**, os processos associados a **eventos/alarmes** e **ações do sistema operativo** sofrem de várias **limitações e exigências temporais**

Para resolver este problema os processos são **agrupados** em grupos de **diferentes prioridades**

- Processos de maior prioridade são executados primeiros
- Processos de menor prioridade podem sofrer `starvation`

Prioridades Estáticas

As prioridades a atribuir a cada processo são determinadas *a priori* de forma **determinística**

- Os processos são **agrupados em classes de prioridade fixa**, de acordo com a sua importância relativa
- Existe risco de os processos menos prioritários sofrerem `starvation`

- Mas se um **processo de baixa prioridade não é executado** é porque o sistema foi **mal dimensionado**
- É o sistema de `scheduling` mais injusto
- É usado em sistemas de tempo real, para garantir que os processos que são críticos são sempre executados

Alternativamente, pode se fazer:

1. Quando um processo é criado, é lhe **atribuído um dado nível de prioridade**
2. Em `time-out` a prioridade do processo é **decrementada**
3. Na ocorrência de um `wait event` a prioridade é **incrementada**
4. Quando o valor de **prioridade atinge um mínimo**, o valor da prioridade sofre um `reset`
 - É colocada no valor inicial, garantindo que o processo é executado

Previnem-se as situações de `starvation` impedindo que o processo não acaba por ficar com uma prioridade tão baixa que nunca mais consegue ganhar acesso

Prioridades Dinâmicas

- As classes de prioridades estão definidas de forma funcional *a priori*
- A mudança de um processo de classe é efetuada com base na utilização última janela de execução temporal que foi atribuída ao processo

Por exemplo:

- **Prioridade 1:** `terminais`
 - Um processo entra nesta categoria quando se efetua a transição `event occurs` (evento de escrita/leitura de um periférico) quando estava à espera de dados do `standard input device`
- **Prioridade 2:** `generic I/O`
 - Um processo entra nesta categoria quando efetua a transição `event occurs` se estava à espera de dados de **outro tipo de input device** que não o `stdin`
- **Prioridade 3:** `small time quantum`
 - Um processo entra nesta classe quando ocorre um `time-out`
- **Prioridade 4:** `large time quantum`
 - Um processo entra nesta classe após um sucessivo número de `time-outs`
 - São claramente processos `CPU-bound` e o objetivo é atribuir-lhes janelas de execução com grande `time quantum`, mas menos vezes

Shortest job first (SJF) / Shortest process next (SPN)

Em sistemas `batch`, o `turnaround time` deve ser minimizado.

Se forem conhecidas **estimativas do tempo de execução** *a priori*, é possível estabelecer uma **ordem de execução** dos processos que **minimizam o tempo de turnaround médio** para um dado grupo de processos

Assumindo que temos N `jobs` e que o tempo de execução de cada um deles é te_n , com $n = 1, 2, \dots, N$. O `average turnaround time` é:

$$t_m = te_1 + \frac{N-1}{N} \cdot te_2 + \dots + \frac{1}{N} \cdot te_N$$

onde t_m é o `turnaround time` mínimo se os `jobs` forem sorteados por ordem ascendente de tempo de execução (estimado)

Para **sistemas interativos**, podemos usar um sistema semelhante:

- Estimamos a taxa de ocupação da próxima janela de execução baseada na taxa de ocupação das janelas temporais passadas
- Atribuímos o processador ao processo cuja estimativa for a **mais baixa**

Considerando fe_1 como sendo a **estimativa da taxa de ocupação** da primeira janela temporal atribuída a um processo e f_1 a fração de tempo efetivamente ocupada:

- A estimativa da segunda fração de tempo necessária é

$$fe_2 = a \cdot fe_1 + (1 - a) \cdot f_1$$

- A estimativa da e-ésima fração de tempo necessária é:

$$fe_N = a \cdot fe_{N-1} + (1 - a) \cdot f_{N-1}$$

Ou alternativamente:

$$a^{N-1} \cdot fe_1 + a^{N-2} \cdot (1 - a) \cdot fe_2 + a \cdot (1 - a) \cdot fe_{N-2} + (1 - a) \cdot fe_{N-1}$$

Com $a \in [0, 1]$, onde a é um coeficiente que representa o peso que a história passada de execução do processo influencia a estimativa do presente

Esta alternativa levanta o problema que que processos `CPU-bound` podem sofrer de `starvation`. Este problema pode ser resolvido contabilizando o tempo que um processo está em espera (`aging`) enquanto está na fila de processos `ready`

Normalizando esse tempo em função do período de execução e denominando-o R , a **prioridade** de um processo pode ser dada por:

$$p = \frac{1 + b \cdot R}{fe_N}$$

onde b é o coeficiente que **controla o peso do aging** na fila de espera dos processos `ready`

4.5 Scheduling Policies

4.5.1 First Come, First Serve (FCFS)

Também conhecido como *First In First Out* (FIFO). O processo mais antigo na fila de espera dos processos *ready* é o primeiro a ser selecionado.

- *Non-preemptive* (em sentido estrito), podendo ser combinado com um esquema de prioridades baixo
- Favorece processos *CPU-bound* em detrimento de processos *I/O-bound*
- Pode resultar num **mau uso** do processador e dos dispositivos de I/O
- Pode ser utilizado com **low priority schemas**

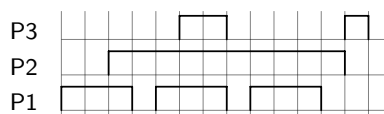


Figure 21: Problema de Scheduling

Usando uma política de *first come first serve*, o resultado do scheduling do processador é:



Figure 22: Política FCFS

- O P1 começa a usar o CPU.
- Como é um sistema FCFS, o processo 1 só larga o CPU passado 3 ciclos.
- O processo P2 é o processo seguinte na fila *ready*, e ocupa o CPU durante 10 ciclos.
- Quando P2 termina, P1 é o processo que está à mais tempo à espera, sendo ele que é executado
- Quando P2 abandona voluntariamente o CPU, o processo P1 corre os seus primeiros dois ciclos
- Quando P3 liberta o CPU, o processo P1 termina os últimos 3 ciclos que precisa
- Quando P3 liberta o CPU, o processo P1 como é *I/O-bound* e precisa de 5 ciclos para o dispositivo estar pronto fica mais dois ciclos à espera para poder terminar executando o seu último ciclo

4.5.2 Round-Robin

- *Preemptive*
 - O *scheduler* efetua a gestão baseado num *clock*
 - A cada processo é atribuído um *time-quantum* máximo antes de ser *preempted*
- O processo **mais antigo** em *ready* é o **primeiro a ser selecionado**
 - não são consideradas prioridades
- Efetivo em sistemas *time sharing* com objetivos globais e sistemas que processem transações

- **Favorece CPU-bound** em detrimento de processos I/O-bound
- Pode resultar num **mau uso de dispositivos I/O**

Na escolha/otimização do `time quantum` existe um **tradeoff**:

- **tempos muito curtos** favorecem a execução de **processos pequenos**
 - estes processos vão ser executados **rapidamente**
- **tempos muito curtos** obrigam a `processing overheads` devido ao `process switching` **intensivo**

Para os processos apresentados acima, o diagrama temporal de utilização do processador, para um `time-quantum` de 3 ciclos é:

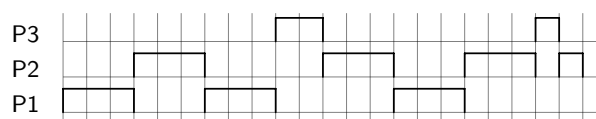


Figure 23: Política Round-Robin

A história de processos em `ready` em fila de espera é: 2, 1, 3, 2, 1, 2, 3, 1

4.5.3 Shortest Process Next (SPN) ou Shortest Job First (SJF)

- `Non-preemptive`
- O process com o `shortest CPU burst time` (menor tempo espectável de utilização do CPU) é o **próximo a ser selecionado**
 - Se vários processos tem o **mesmo tempo de execução** é usado FCFS para desempatar
- Existe um **risco de starvation** para grandes processos
 - o seu acesso ao CPU pode ser **sucessivamente adiado** se existir “forem existindo” processos com **tempo de execução menor**
- Normalmente é usado em escalonamento de logo prazo, `long-term scheduling` em sistemas `batch`, porque os utilizadores esperam estimar com precisão o tempo máximo que o processo necessita para ser executado

4.5.4 Linux

No Linux existem 3 classes de prioridades:

1. **FIFO**, `SCHED_FIFO`
 - `real-time threads`, com política de prioridades
 - uma `thread` em execução é `preempted` apenas se um processo de **mais alta prioridade da mesma classe** transita para o estado `ready`

- uma `thread` em execução pode **voluntariamente abandonar o processador**, executando a primitiva `sched_yield`
- dentro da mesma classe de prioridade a política escolhida é `First Come, First Serve` (FCFS)
- Só o `root` é que pode lançar processos em modo FIFO

2. Round-Robin real time threads, `SCHED_RR`

- `threads` com prioridades com necessidades de execução em tempo real
- Processos nesta classe de prioridades são `preempted` se o seu `time-quantum` termina

3. Non real time threads, `SCHED_OTHER`

- Só são executadas se não existir nenhuma `thread` com necessidades de execução em tempo real
- Está associada à processos do utilizador
- A política de escalonamento tem mudado à medida que a são lançadas novas versões do `kernel`

A **escala de prioridades** varia

- 0 a 99 para `real-time threads`
- 100 a 139 para as restantes

Para lançar uma `thread` (sem necessidades de execução em tempo real) com diferentes prioridades, pode ser usado comando `nice`.

Por *default*, o comando lança uma `thread` com prioridade 120. O comando aceita um `offset` de [-20, +19] para obter a prioridade mínima ou máxima.

Algoritmo Tradicional

- Na classe `SCHED_OTHER` as prioridades são baseadas em **créditos**
- Os créditos do processo em execução são **decrementados** à medida que ocorre uma interrupção do `real time clock`
- O processo é `preempted` quando são atingidos zero créditos
- Quando todos os processos `ready` têm zero créditos, os créditos de **todos os processos** (incluindo os que estão bloqueados) são **recalculados** segundo a fórmula:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} + PBase_j + nice_j$$

onde são tido em conta a **história passada de execução do processo** e as **prioridades**

- O `response time` de processos `I/O-bound` é minimizado
- A `starvation` de processos `CPU-bound` é evitada
- Solução **não adequada para múltiplos processadores** e é má se o número de processos é elevado

4.6 Novo Algoritmo

- Os processos na classe `SCHED_OTHER` passam a usar um `completely fair scheduler` (CFS)
- O scheduling é baseado no `vruntime`, *virtual run time*, que mede durante quanto tempo uma `thread` esteve em execução

- o `virtual run time` está relacionado quer com o **tempo de execução real** (`physical run time`) e a **prioridade** da `thread`
- Quanto maior a prioridade de um processo, menor o `physical run time`
- O `scheduler` seleciona as `threads` com menor `virtual run time`
 - Uma `thread` com prioridade mais elevada que fique pronta a ser executada pode “forçar” um `preempt` uma `thread` com menor prioridade
 - * Assim é possível que uma `thread I/O bound` “forçar” o processador a `preempt` um processo `CPU-bound`
- O algoritmo é implementado com base numa `red-black tree` do processador