Execução Concorrente de Processos

- Os processos de um SO podem executar concorrentemente, partilhando o CPU num dado intervalo de tempo.
- É o **temporizador** (ou **scheduler**), um programa do SO, quem distribui o tempo de CPU pelos vários processos prontos a executar.
- Vantagens da execução concorrente:
  - **partilha de recursos físicos e lógicos** – por múltiplos utilizadores.
  - **maior eficiência e modularidade** – podemos ter várias tarefas em simultãneo, e num sistema multiprocessador executá-las mais rápido.
- Contudo, a execução concorrente de processos que cooperam entre si, requer a existência de mecanismos de sincronização e comunicação.

Comunicação entre processos (IPC)

Métodos de comunicação entre processos:

- **Sinais unidireccionais:**
  - um processo-filho pode enviar sinais, através de `exit()/return()` ao processo-pai que pode fazer a recepção com `wait()`.
  - um processo pode enviar a outro (desde que relacionados) um sinal explícito, através de `kill()`. O processo que recebe o sinal deve executar um `signal()`.
- **Pipes:** um processo escreve e outro lê (unidireccional).
- **Mensagens:** um processo envia uma mensagem explícita a outro.
- **Partilha de Memória:** dois processos que partilhem uma variável/ficheiro, podem comunicar entre si escrevendo e lendo dessa variável/ficheiro.

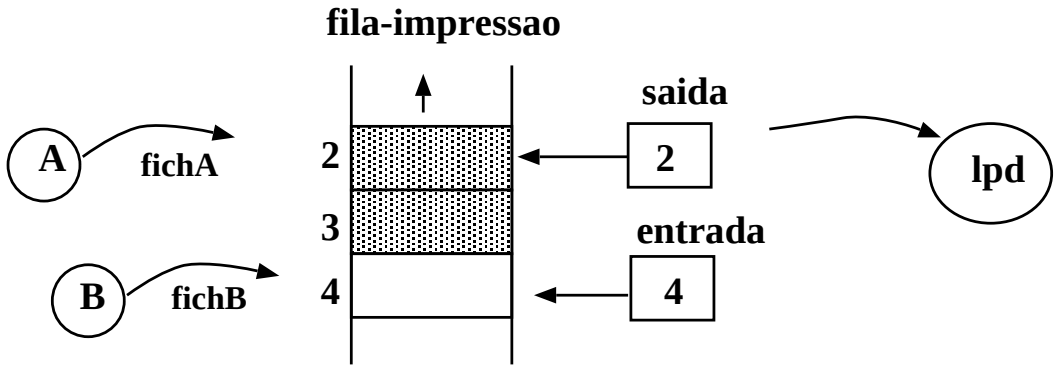
Competição entre processos (race conditions)

Existe *competição entre processos* quando o resultado depende da ordem de execução dos processos, e.g. dois processos a alterarem algo partilhado por ambos.

Exemplo: Imprimir um ficheiro

Quando um processo imprime um ficheiro, o nome deste é colocado numa fila de impressão (spool directory) e é o programa `lpd` (printer-daemon) que se encarrega de periodicamente verificar se há ficheiros na fila e se houver, imprime-os e remove os nomes respectivos da fila.

Suponhamos que dois processos A e B, quase simultaneamente, decidem imprimir um ficheiro cada. A figura ilustra a fila de execução actual:

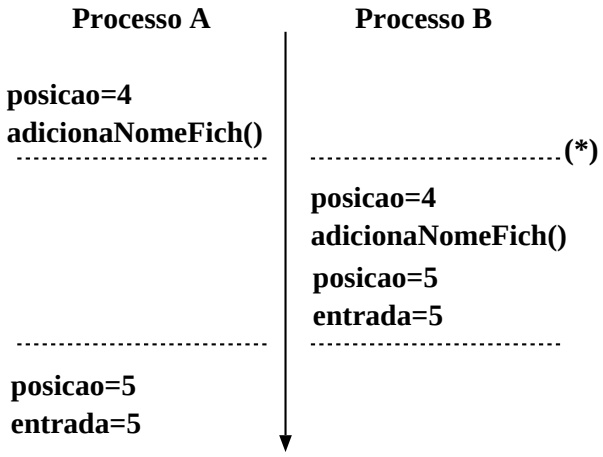


suponha que o código a executar pelos 2 processos para adicionar os ficheiros à fila é:

```
posição= entrada
adicionaNomeFila(nomeFich, posição)
posição++
entrada= posição
```

Exemplo de competição entre processos

Suponhamos que tinhamos o seguinte padrão de execução:



(*) Proc. A interrompido pelo scheduler.

A fila foi actualizada como se tivessemos adicionado apenas um ficheiro, assim o sugere a variável `entrada`. O ficheiro que o processo `A` pretendia imprimir perde-se.

→ É necessário ter atenção à actualização concorrente da variável `entrada` pelos dois processos.

Zonas Críticas

- n processos a competirem para acederem a variáveis partilhadas.
- cada processo tem uma parte de código, *zona crítica*, na qual acede a memória partilhada.
- **Problema:** assegurar que quando um processo está a executar a sua zona crítica, nenhum outro processo pode executar na sua zona crítica.

  Se apenas permitirmos um processo de cada vez na zona crítica, evita-se competição entre processos.
- Estrutura do processo P_i:

```
repeat
  entrar_zc
    zona crítica
  sair_zc
    zona restante de código
until false;
```

Como evitar competição entre processos em zonas críticas?

1. **Exclusão Mútua**: nas zonas críticas não poderão estar nunca 2 processos em simultâneo.
2. nenhum processo deverá ter de esperar eternamente para entrar na sua zona crítica.
3. nenhum processo que esteja fora da sua zona crítica, poderá bloquear outros processos.
4. não se pode assumir velocidade ou número de CPUs.

Métodos de exclusão mútua com espera activa:

1. Desligar interrupções (solução hardware):

```
desligar_interrupções
 zona crítica
ligar_interrupções
```

- com as interrupções desligadas, o CPU não poderá ser comutado para outro processo.
- *método útil a nível do kernel*, o scheduler usa-o, mas não é apropriado como mecanismo geral para garantir exclusão mútua entre processos-utilizador.
- se um processo-utilizador pudesse desligar interrupções, poderia ser o fim do sistema. Porquê?

2. Variáveis de Bloqueio

- Variáveis partilhadas:
  – **boolean** `flag[2]`;
  – inicialmente `flag[0] = flag[1] = false;`
  – `flag[j] == false` $\Rightarrow P_i$ pode entrar na zona crítica.

```
Processo 0:                          Processo 1:
...                                  ...
   while (flag[1]) ;                    while (flag[0]) ;
   flag[0]= true;                       flag[1]= true;
     zona_crítica();                      zona_crítica();
   flag[0]= false;                      flag[1]= false;
     zona_não_crítica();                  zona_não_crítica();
...                                  ...
```

- Esta solução não satisfaz exclusão mútua! Porquê?

  P_0 executa o ciclo-**while** e encontra `flag[1]=false`;
  P_1 executa o ciclo-**while** e encontra `flag[0]=false`;
  P_0 executa `flag[0]=true` e entra na `zona_critica()`;
  P_1 executa `flag[1]=true` e entra na `zona_critica()`;

  Depende da ordem de execução dos processos.

3. Alternância estrita

- Variáveis partilhadas:
  – **int** `vez`; inicialmente `vez = 0`
  – `vez == i` $\Rightarrow P_i$ pode entrar na zona crítica.

```
Processo 0:                          Processo 1:
...                                  ...
   while (vez!=0) ;                    while (vez!=1) ;
     zona_crítica();                     zona_crítica();
   vez= 1;                             vez= 0;
     zona_não_crítica();                 zona_não_crítica();
...                                  ...
```

- Esta solução satisfaz exclusão mútua, mas desperdiça CPU (a não ser que o tempo de espera seja curto).
- Só funciona se houver alternância de vez entre dois processos.
- Se um dos processos falhar o outro fica bloqueado.
- **espera activa** – teste contínuo de uma variável à espera que ela tome um dado valor (**while (vez!=0) ;**).

4. Algoritmo de Peterson

- Em 1965, Dijkstra apresentou um algoritmo que garantia exclusão mútua de dois processos, desenvolvido pelo matemático holandês Dekker.
- Dekker combinou a ideia de alternar vez com a ideia de variáveis de bloqueio e variáveis de aviso.
- Em 1981, G.L. Peterson propôs uma solução mais simples: combina alternância estrita com uma outra variável que indica se o processo está ou não interessado em entrar na zona crítica.

– **boolean** `flag[2]`;

– **int** `vez`;

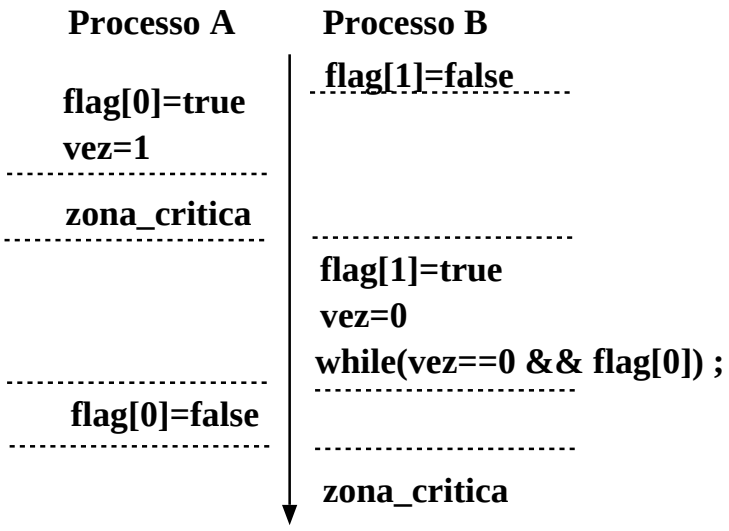
– `(flag[j]==false || vez=i)` $\Rightarrow P_i$ pode entrar entrar na zona crítica.

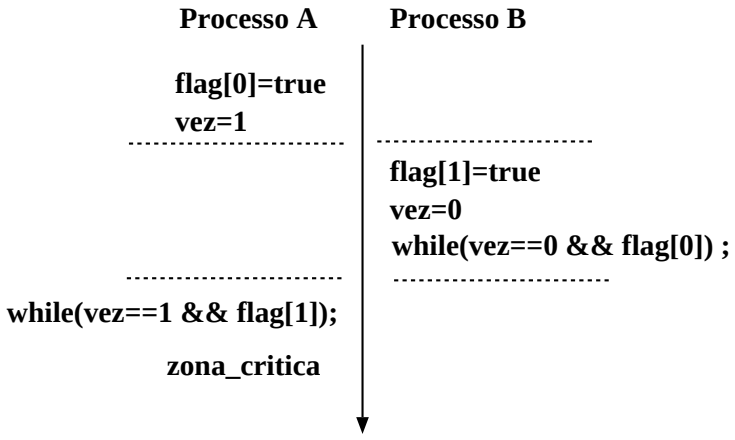
```
Processo 0:
...
  flag[0]=true;
  vez= 1;
  while (vez==1 && flag[1]) ;
    zona_crítica();
  flag[0]=false;
    zona_não_crítica();
...
```

```
Processo 1:
...
   flag[1]=true;
   vez= 0;
    while (vez==0 && flag[0]) ;
      zona_crítica();
   flag[1]= false;
     zona_não_crítica();
...
```

Algoritmo de Peterson em funcionamento

Dois exemplos do funcionamento do algoritmo:





O processo que executar `vez=valor` em último, fica sem conseguir entrar na zona crítica.

Instrução "Test and Set" (Lock)

- Testa e modifica o conteúdo de uma posição de memória de forma atómica.
- A instrução corresponde à função:

```
int TSL(int *m) {
  int r;

  r= *m;
  *m= 1;
  return r;
}
```

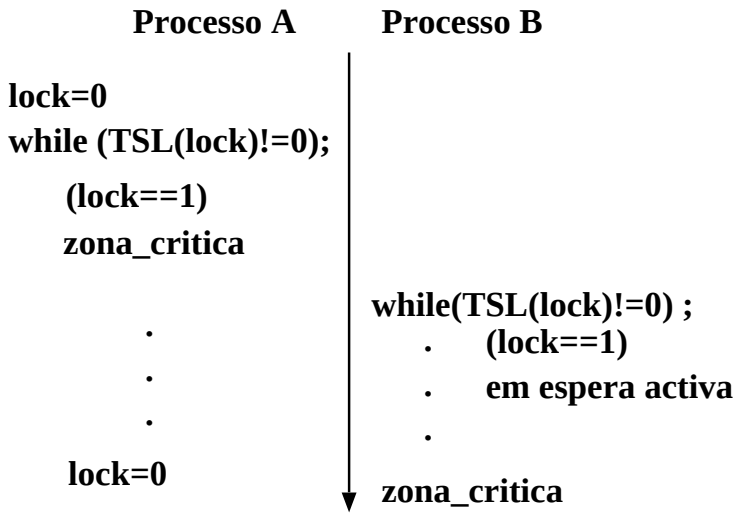
- a execução da função `TSL(m)` tem de ser indivisível, i.e. nenhum outro processo pode aceder à posição de memória `m` até que a instrução tenha sido executada.
- Como usar a instrução `TSL()` de forma a garantir exclusão mútua no acesso a uma zona crítica?

– usar uma variável partilhada `lock` que qualquer processo possa modificar;
– obrigar um processo a activar o `lock` antes de entrar na zona crítica;
– usar `TSL()` para conseguir modificar `lock` de forma atómica).

Exclusão mútua com Test-and-Set

- Variável partilhada:
  – **int** `lock`; inicialmente `lock=0`
  – se `TSL(lock)==0` $\Rightarrow$ P_i pode aceder à zona crítica.
- Algoritmo de P_i:

```
...
    while (TSL(&lock)!=0) ;  % espera activa
       zona_critica();
    lock= 0;
       zona_não_crítica();
...
```



- Vantagens e inconvenientes:
  – pode ser usada por um número arbitrário de processos.
  – é simples e fácil de verificar.
  – pode suportar zonas críticas múltiplas.
  – com número maior de processos, espera activa pode ser problema.
  – é possível os processos entrarem em "starvation".

Semáforos (Dijkstra, 1965)

- permite sincronizar processos no acesso a zonas críticas, e não envolve espera activa.
- um semáforo é definido como um inteiro não-negativo, ao qual estão associadas duas operações atómicas (indivisiveis):

```
down(S) ou wait(S):
-------------------
  if (S==0)
    suspende execução do processo
  S--;

up(S) ou signal(S):
-------------------
  S++;
  if (S==1)
    retoma um processo suspenso em S
```

- *Semáforos Binários*: apenas tomam valores 0 e 1. São habitualmente usados para garantir exclusão mútua no acesso a zonas críticas, e designam-se por `mutexs`.

Exclusão mútua com semáforos

- Variáveis partilhadas:
  – **semáforo** `mutex`; inicialmente `mutex=1`.
- Processo P_i:

```
...
    down(mutex);
       zona_crítica();
    up(mutex);
       zona_não_crítica();
...
```

- O processo consegue aceder à zona crítica se o `mutex= 1` quando executou `wait(mutex)`. Se estivesse `mutex=0`, então o processo adormecia à espera que alguém (que está na zona crítica) sinalize o `mutex`.

Problema do Produtor/Consumidor

Consideremos dois processos que partilham um `buffer` com capacidade para `N` elementos.

Um processo, produtor, coloca informação no depósito, enquanto outro processo, o consumidor, retira informação do depósito.

Problemas que podem surgir:

- produtor quer adicionar um ítem, mas o depósito está cheio.

- consumidor quer retirar um ítem, mas o depósito está vazio.

Solução do Produtor/Consumidor com Semáforos

```
typedef int semaforo;

semaforo mutex= 1; /* para garantir exclusão mútua  */
semaforo vazio= N; /* num. posições vazias no buffer*/
semaforo cheio= 0; /* num. posições cheias no buffer*/
```

```
produtor() {
  int item;

  while (True) {
    produz(&item);
    down(&vazio);
    down(&mutex);
    adiciona(item);
    up(&mutex);
    up(&cheio);
  }
}
```

```
consumidor() {
  int item;

  while (True) {
    down(&cheio);
    down(&mutex);
    retira(item);
    up(&mutex);
    up(&vazio);
    consome(item);
  }
}
```

Os semáforos `vazio` e `cheio` são usados para sincronizar os dois processos, permitindo-lhes suspender caso a operação que pretendem realizar não possa prosseguir.

Implementação de Semáforos

- Um semáforo é definido por uma estrutura com dois campos:

```
typedef struct {
         int val;
  ProcessList *L;
} Semaforo;
```

- Assume-se duas operações simples:
  – *block()* suspende o processo que a invoca;
  – *wakeup(P)* retoma a execução do processo suspenso P.
- As operações sobre os semáforos:

```
Semaforo S;

down(S):  if (S.val==0) {
            adiciona(processID, S.L);
            block();
          }
          S.val--;

up(S):    S.val++;
          if (S.val==1) {
            pid_susp= retira_primeiro(S.L);
            wakeup(pid_susp);
          }
```

Implementação de Semáforos com a instrução TSL

• usa-se espera activa em vez de lista de processos.

```
typedef enum {False,True} BOOL;
typedef struct {
   int val;
  BOOL mutex;
  BOOL espera;
} Semaforo;
Semaforo S={1,False,True};

#define DOWN(S) {              \
  while (TSL(&S.mutex));       \
  if (S.val==0) {              \
    S.mutex=False;             \
    while (TSL(&S.espera)) ;   \
  }                            \
  S.val--;                     \
  S.mutex=False;               \
}

#define UP(S) {                \
  while (TSL(&S.mutex));       \
  S.val++;                     \
  if (S.val == 1) {            \
    while (!S.espera) ;        \
    S.espera=False;            \
  }                            \
  S.mutex=False;               \
}
```

Encravamento (Deadlock) e Inanição (Starvation)

- Atenção à ordem de chamada das operações sobre um semáforo! Podemos facilmente criar uma situação favorável a encravamentos.
- **encravamento** (deadlock) – verifica-se quando dois ou mais processos ficam à espera pela ocorrência de um evento que só pode ser causado por um dos processos em espera.
- Exemplo: sejam S e Q dois semáforos inicializados em 1,

```
   P0                P1
down(S)           down(Q)
down(Q)           down(S)
 ...               ...
up(S)             up(Q)
up(Q)             up(S)
```

- **Inanição** (starvation) – verifica-se quando um processo fica à espera de vez de acesso a um semáforo por tempo indefinido. O processo está em execução mas não consegue acesso ao recurso.

Monitor (Hoare 1974 e Hansen 1975)

- primitiva de alto-nível para sincronização de processos concorrentes no acesso a recursos partilhados.
- é um tipo-abstracto de dados, constituído por variáveis, estruturas de dados e procedimentos que operam sobre essas variáveis e estruturas de dados.
- um programa apenas tem acesso aos procedimentos do monitor.
- goza da seguinte propriedade:

  em qualquer instante, apenas pode estar um processo activo dentro do monitor.

  que garante exclusão mútua.
- se um processo invoca um procedimento do monitor (i.e. "pretende entrar no monitor"), e existir outro processo activo dentro do monitor, é suspenso e colocado numa fila de espera, à entrada, até que o outro processo deixe o monitor.
- os procedimentos incluem em si as zonas críticas de código nas quais se pretende garantir exclusão mútua.
- nas zonas críticas, e quando o processo não puder continuar a executar parte do código, interessa-nos permitir que um processo possa suspender a execução numa condição:

  → *variáveis de condição* + operações *cwait()* e *csignal()*.

Monitores: variáveis de condição + cwait() e csignal()

- as variáveis de condição são definidas por (depende da linguagem!):

  `CondVar x;` – `x` é variável de condição.

- *cwait(x)*: o processo que executa a operação suspende na variável de condição `x`, até que outro processo sinalize esta variável.

- *csignal(x)*: o processo que executa a operação acorda um dos processos suspensos (normalmente o primeiro da fila) nesta variável de condição.

- Quando um processo suspende dentro do monitor por acção de *cwait()*, o monitor fica livre para acesso por outro processo.

- como *csignal(x)* acorda um dos processos suspensos em x, como evitar que estejam dois processos activos dentro do monitor. Duas estratégias:

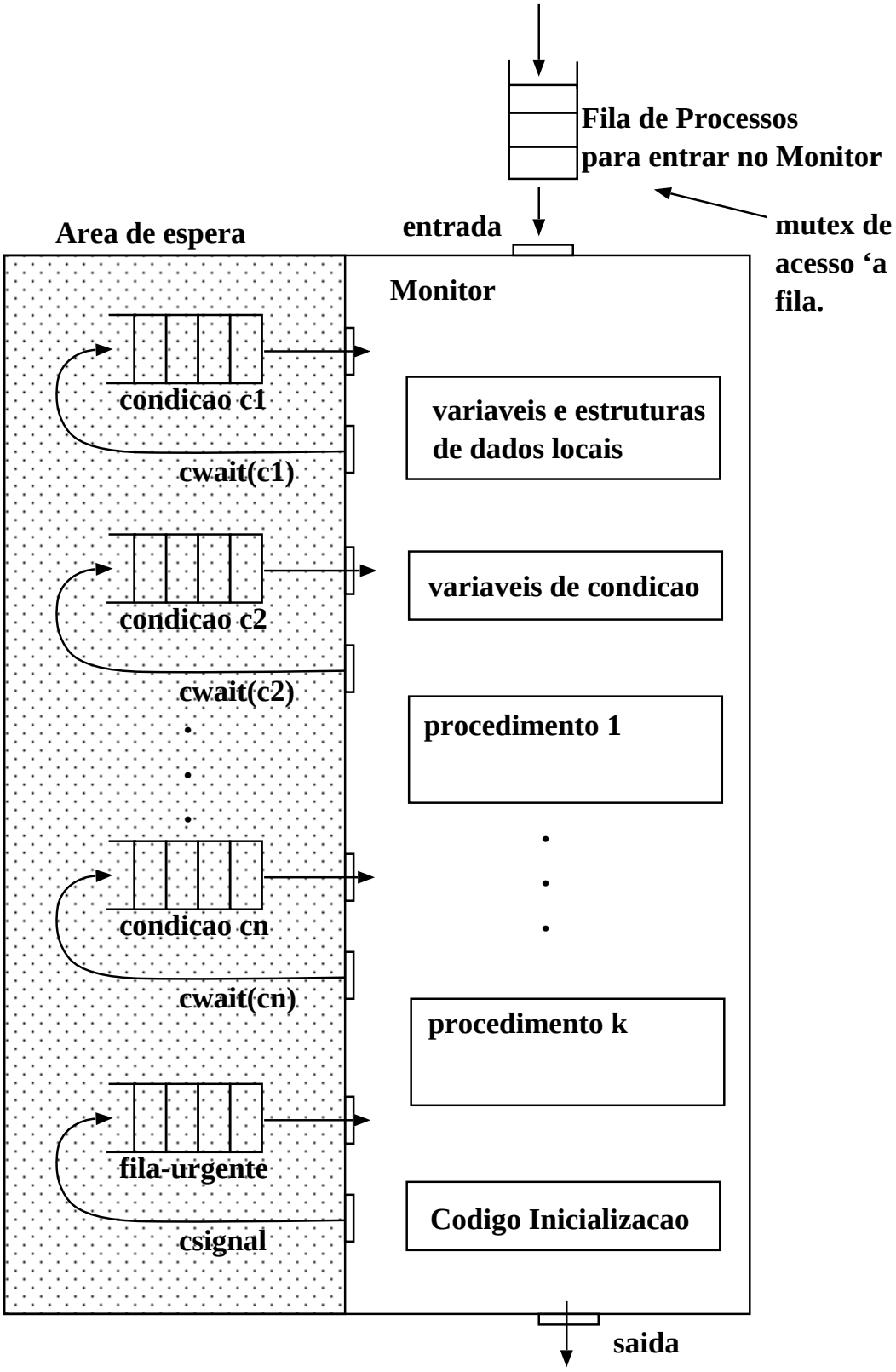
  – (*Hansen*) o processo que faz o *csignal()* deixa de imediato o monitor.
  $\rightarrow$ *csignal()* é a última instrução do procedimento!

  – (*Hoare*) o processo que foi acordado deve executar de imediato, suspendendo-se o processo que fez o *csignal()*.

  A solução de Hansen é a mais simples de concretizar e normalmente é a usada.

- as variáveis de condição não acumulam os sinais.

Estrutura de um monitor



Exemplo com monitores: Produtor/Consumidor

```
MONITOR  PC {
  int buf[N],first,last;    /*buffer e vars. de acesso*/
  int ctr;                  /*num. elementos no buffer*/
  CondVar naoCheio,naoVazio;/*vars. de condição      */

  void adicionar(int val) {
    if (ctr==N)           /* buffer-cheio ?          */
      cwait(naoCheio);
    buf[first]= val;      /* adiciona valor a buffer */
    first= (first+1) % N;
    ctr++;
    if (ctr==1)           /* deixou de estar vazio ? */
      csignal(naoVazio);
  }

  int retirar() {
    int val;

    if (ctr==0)           /* buffer-vazio ?          */
      cwait(naoVazio);
    val=buf[last];        /* retira valor do buffer  */
    last= (last+1) % N;
    ctr--;
    if (ctr==(N-1))       /* deixou de estar cheio ? */
      csignal(naoCheio);
    return val;
  }

  void init() {
    first=last=ctr= 0;
  }
} /* FimMonitor */
```

Exemplo com monitores: (cont.)

```
produtor() {
 int v;

 while (true) {
   produz_item(&v);
   PC.adicionar(v);
 }
}

main() {
  PC.init();
  if (fork()==0)
    produtor();
  else
    consumidor();
}
```

```
consumidor() {
  int v;

  while (true) {
    v= PC.retirar();
    consome_item(v);
  }
}
```

- um dos problemas com monitores é que poucas linguagens oferecem esta primitiva.
- contudo, é possível implementá-la usando *mutexs* ou *semáforos*.
- outro problema é que não funciona para sistemas distribuídos, pois requer memória partilhada na sua implementação.

Troca de Mensagens

• método de comunicação e sincronização entre processos, pode ser usado em sistemas de memória partilhada ou distribuída.

• baseia-se em duas operações:

- *send(destino, mensagem)*:
  – envia a *mensagem* para o processo *destino*. Este processo pode esperar ou não esperar que haja um processo pronto a receber.
- *receive(origem, mensagem)*:
  – recebe uma mensagem previamente enviada pelo processo *origem*. Se não existir mensagem em *origem*, o processo ou espera que a mensagem chegue ou prossegue e ignora o *receive*.

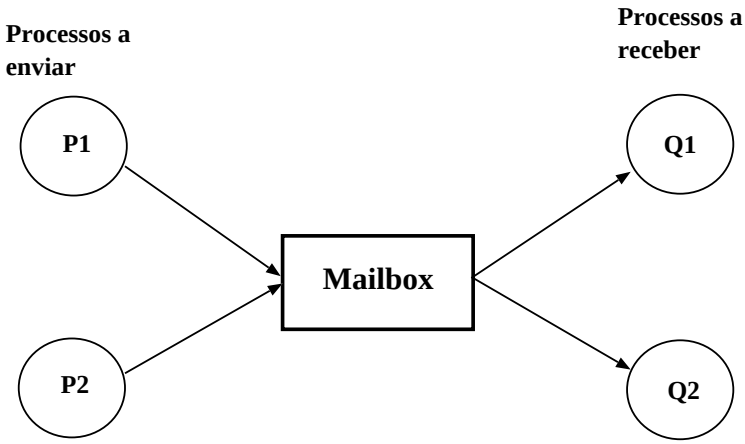
O que conduz às seguintes formas de sincronização:

- **envio com bloqueio; recepção com bloqueio**: ambos emissor e receptor bloqueiam até que consigam sincronizar para o envio/recepção da mensagem. Estratégia designada por *rendez-vous*.
- **envio sem bloqueio; recepção com bloqueio**: apenas o receptor bloqueia à espera de receber a mensagem. É o método mais útil, pois o envio é mais rápido.
- **envio sem bloqueio; recepção sem bloqueio**: nenhuma das partes espera.

Implementação de Troca de Mensagens

Duas formas possíveis, associadas ao tipo de endereçamento dos processos:

- **endereçamento-directo** – os endereços dos processos destino e origem são conhecidos e fixos à partida. Útil em sistemas concorrentes.
- **endereçamento-indirecto** – usa-se uma estrutura de dados intermédia, em memória partilhada, conhecida dos dois processos e através da qual enviam e recebem mensagens. Um exemplo típico são as *caixas-de-correio* (mailboxs).



• uma relação muitos-um (muitos a enviar e um a receber) é útil para interacção do tipo cliente/servidor. Neste caso a mailbox é designada por *porta* (port).

• relação um-muitos é útil para *broadcast*.

Exclusão mútua com troca de mensagens

Considere um conjunto de processos $P_1 \ldots P_n$ e uma mailbox, `mutex`, partilhada pelos processos.

A mailbox é inicializada com uma mensagem de conteúdo vazio.

Um processo para entrar na zona crítica, tenta primeiro receber uma mensagem. Se a mailbox estiver vazia o processo bloqueia. Após conseguir receber a mensagem, executa a zona crítica e envia a mensagem nula para a mailbox.

```
...
receive(mutex, msg);
zona_crítica();
send(mutex,msg);
zona_não_crítica();
...
```

A mensagem funciona como um testemunho que passa de processo para processo, e só quem tiver o testemunho é que entra na zona crítica.

Pressupostos desta solução:

• se existir uma mensagem na mailbox, ela é entregue a apenas um processo enquanto os outros ficam bloqueados.

• se a mailbox estiver vazia, todos os processos ficam bloqueados; quando chega uma mensagem, apenas um processo é activado e recebe a mensagem.

Exemplo de Troca de Mensagens: Produtor/Consumidor

```
produtor() {
...
  while(1) {
    receive(podeproduzir,pmsg);
    pmsg= produz();
    send(podeconsumir,pmsg);
  }
}

consumidor() {
...
  while(1) {
    receive(podeconsumir,cmsg);
    consumir(cmsg);
    send(podeproduzir,null);
  }
}

main() {
...
 create_mailbox(podeproduzir);
 create_mailbox(podeconsumir);
 for (i=0; i<N; i++)
  send(podeproduzir,null);
 if (fork()==0)
  produtor();
 else
  consumidor();
}
```

Observações

• A forma mais geral de comunicação entre processos é através de memória partilhada.

• Os processos têm normalmente um espaço de endereçamento distinto, é no entanto possível definir zonas de memória comuns a dois ou mais processos, como já visto anteriormente.

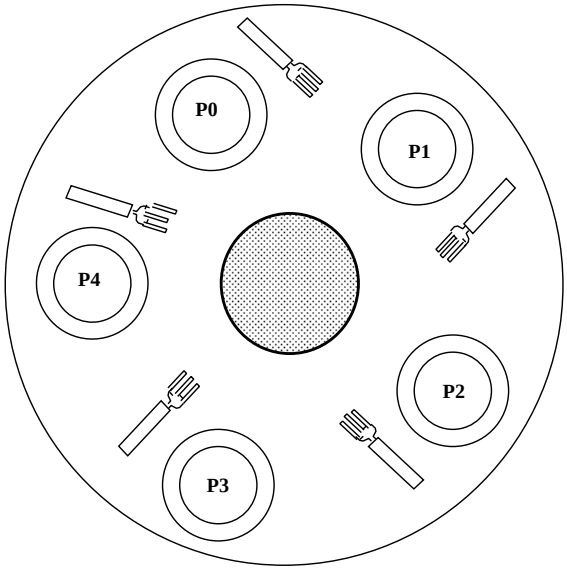
• A vantagem da memória partilhada é que podemos definir variáveis ou estruturas de dados nesse espaço e fazer uso como se de variáveis locais se tratasse.

• O inconveniente é que a sincronização entre os processos, com vista a garantir exclusão mútua, tem de ser feita por um dos métodos vistos antes (test-and-set, semáforos, etc.).

• As trocas de mensagens com recurso a *mailboxs* é algo que se situa, em termos de funcionalidade, entre as *pipes* e a memória partilhada.

Problemas clássicos de IPC: O Jantar dos Filósofos

O problema deve-se a (Dijkstra 1965) e modela processos em competição para acesso exclusivo a um número limitado de recursos.



Descrição:

– 5 filósofos sentados a uma mesa;

– cada filósofo tem um prato com esparguete;

– para comer, um filósofo precisa de dois garfos;

– entre dois pratos existe apenas um garfo (no. garfos = no. filósofos).

A vida de um filósofo consiste em períodos alternados de **comer** e **pensar**. Quando tem fome, tenta obter o garfo esquerdo e depois o direito, um de cada vez. Caso consiga os dois garfos, come durante algum tempo, depois pousa os garfos e continua a pensar.

Será possível escrever um programa que simule o comportamento dos filósofos (processos concorrentes) sem deixar que cheguem a uma situação de deadlock ou starvation?

deadlock – todos pegam em um garfo e ficam à espera de conseguir o segundo!

starvation – pegar e largar o garfo sem nunca conseguir os dois (os processos estão em execução, mas não conseguem aceder aos recursos).

Solução para o Jantar dos Filósofos

A execução de um filósofo poderia corresponder a:

```
pensar();
pegar_garfo(dir);
pegar_garfo(esq);
comer();
pousar_garfo(esq);
pousar_garfo(dir);
```

contudo esta solução não funciona, porque permite situações em que todos os filósofos peguem no garfo direito ao mesmo tempo, ficando depois à espera de conseguirem o garfo esquerdo.

→ proteger o acesso à zona crítica (i.e pegar nos dois garfos).

• se usarmos apenas um *mutex*, resolvemos o problema mas ficamos com os filósofos a comer à vez. Alternativas:

- associar um semáforo por filósofo, permitindo-lhe suspender caso não consiga os dois garfos e esperar ser acordado por outros.
- associar um estado (pensar=0, fome=1, comer=2) a cada filósofo;

assim, um filósofo só consegue os dois garfos, se:

- estiver com fome, e se
- o filósofo da esquerda e da direita não estiverem a comer.

Jantar dos Filósofos usando Semáforos

```
Semáforo mutex=1; /* controla acesso zona_crítica */
Semáforo s[N];    /* um semáforo por filósofo     */
int estado[N];    /* PENSAR=0, FOME=1, COMER=2    *
filosofo(int i) {
  while(True) {
    pensar();
    pegar_garfos(i);
    comer();
    pousar_garfos(i);
  }
}
pegar_garfos(int i) {
  DOWN(&mutex);       /* entrar na zona crítica   */
  estado[i]=FOME;
  tentativa(i);       /* tenta apanhar 2 garfos   */
  UP(&mutex);
  DOWN(&s[i]);        /* bloqueia se não conseguir*/
}
pousar_garfos(int i) {
  DOWN(&mutex);
  estado[i]=PENSAR;
  tentativa(ESQ);     /* vizinho ESQ está comer ? */
  tentativa(DIR);     /* vizinho DIR está comer ? */
  UP(&mutex);
}
tentativa(int i) {
  if (estado[i]==FOME && estado[ESQ]!=COMER
      && estado[DIR]!=COMER) {
    estado[i]=COMER;
    UP(&S[i]);
  }
}
main() { int i;
  for (i=0;i<N;i++) if (fork()==0) {filósofo(i);exit();}
}
```

Problemas clássicos de IPC: Leitores e Escritores

→ modela o acesso a uma base de dados. Ex: sistema de reserva de passagens aéreas.

- vários leitores a acederem em simultãneo;
- apenas um escritor pode estar a actualizar a base de dados, sem que qualquer outro processo, escritor ou leitor, tenha acesso.

Soluções:

- *prioridade aos leitores*. Se um escritor quiser actualizar a BD e existirem leitores a aceder, o escritor espera!
- *prioridade dos escritores*. enquanto houver um escritor que queira ou esteja a actualizar a BD, nenhum leitor pode conseguir o acesso.
- *Qualquer das soluções pode conduzir a starvation!*.

```
Semaforo mutex;
Semaforo  bd;
int numLeit;

escritor() {
...
  down(bd);
  ...
  escreve();
  ...
  up(bd);
}
```

```
leitor() {
...
  down(mutex),
    numLeit++;
    if (numLeit==1)
    down(bd);
  up(mutex);
  le();
  down(mutex);
    numLeit--;
    if (numLeit==0)
    up(bd);
  up(mutex);
}
```

Problemas clássicos de IPC: o Barbeiro dorminhoco.

- 1 barbeiro; 1 cadeira de barbeiro
- N cadeiras para os clientes esperarem.

Se não existirem clientes, o barbeiro senta-se e dorme.
Quando chega um cliente, acorda o barbeiro. Se chegarem mais clientes enquanto o barbeiro estiver a cortar um cabelo, sentam-se, caso tenham cadeiras livres, ou deixam a barbearia (se as cadeiras estiverem ocupadas).

Como evitar competição entre os processos cliente e barbeiro?

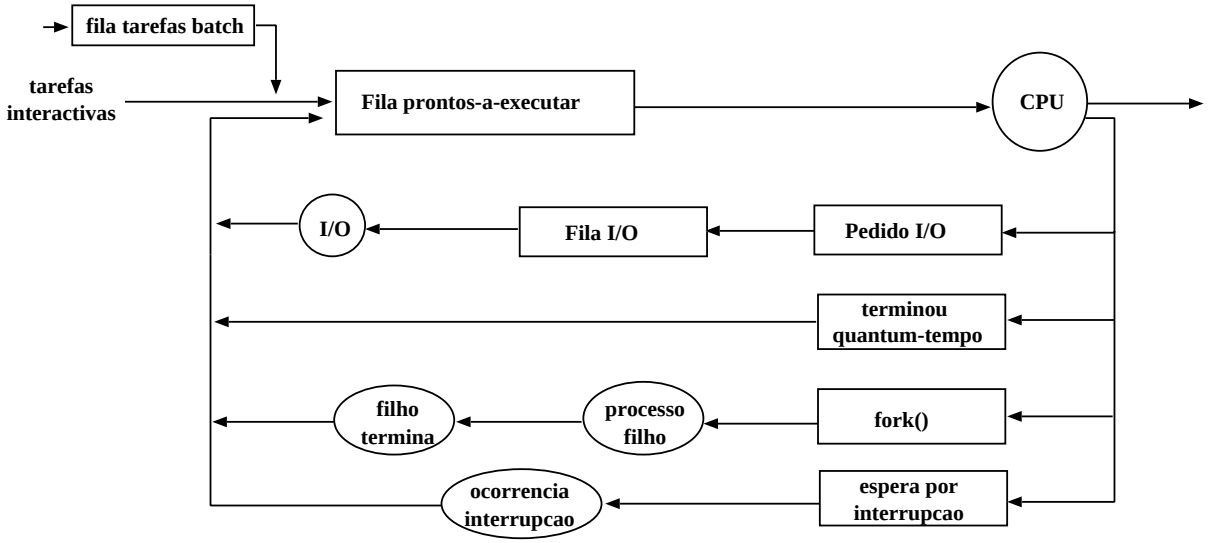
```
Semaforo clientes=0; /* #clientes à espera de vez      */
Semaforo barbeiros=0;/* #barbeiros à espera de clientes*/
Semaforo mutex=1;    /* sem. binário exclusão mútua    */
int sentados=0;      /* #clientes sentados             */

barbeiros() {
  while(1) {
    down(clientes);  /*existem clientes? se não adormece*/
    down(mutex);
    sentados--;      /*menos um cliente à espera        */
    up(barbeiros);   /*menos um barbeiro adormecido     */
    up(mutex);
    cortar(); }
}
clientes() {
  down(mutex);         /*se não existem cadeiras livres*/
  if (sentados<NCads) {/*vai embora; se existem entra  */
    sentados++;        /*mais um cliente à espera      */
    up(clientes);      /*acorda barbeiro se necessário */
    up(mutex);         /*liberta zona crítica          */
    down(barbeiros);   /*adormece se não há barbeiros  */
    sentar_e_cortar(); /*livres                        */
  } else
    up(mutex);
}
```

Temporização (Scheduling) de Processos

→ Tem por objectivo maximizar o uso do CPU, i.e. ter sempre um processo a executar.

Filas de processos usadas em scheduling:

- *Fila de tarefas*: processos submetidos para execução, à espera de serem carregados para memória. Podemos separar *tarefas batch* de outras.
- *Fila dos prontos-a-executar*: processos já em memória, prontos para executar.
- *Filas de acesso-a-periféricos*: cada periférico tem uma fila de processos à espera de vez de acesso.
- Um processo migra entre as varias filas.



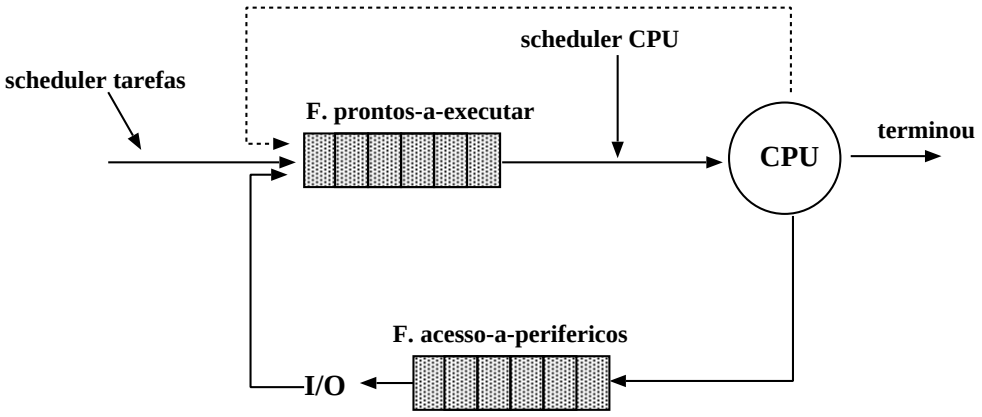
Schedulers (temporizadores)

- scheduler de tarefas (longa-duração): selecciona quais os processos que devem ser carregados para a fila dos prontos-a-executar.

  Activado menos frequentemente (segundos, minutos); rapidez não é crucial.

- scheduler do CPU (curta-duração): selecciona o processo que irá ser executado a seguir e atribui-lhe o CPU.

  Activado com muita frequência (milisegundos), tem de ser muito rápido.



Scheduler de CPU

- Entre os processos que estão em memória prontos-a-executar, selecciona um e atribui-lhe o CPU.
- O scheduler de CPU toma decisões quando um processo:
  1. passa do estado em-execução é suspenso (estado em-espera).
  2. passa do estado em-execução para pronto-a-executar.
  3. passa do estado em-espera para pronto-a-executar.
  4. termina
- Um scheduler diz-se *não-interruptível* (*non-preemptive*) se apenas intervém nas situações 1. e 4.
- Caso contrário, o scheduler diz-se *interruptível* (*preemptive*).

- **Dispatcher**: módulo que dá o controlo do CPU ao processo seleccionado pelo scheduler; passos envolvidos:
  - troca de contexto de processos
  - passar para modo-utilizador
  - re-iniciar a execução do programa
- *Dispatch Latency* - tempo que o *dispatcher* demora para parar um processo e iniciar outro.

Características de um bom algoritmo de scheduling.

- **Uso Eficiente de CPU** – manter o CPU o mais ocupado possível.

  40% = levemente ocupado; 90% = fortemente ocupado.

- **throughput** – nºde processos que terminam a sua execução por unidade de tempo.

- **tempo de turnaround** – qtd. tempo para executar um processo em particular (inclui tempo na fila dos prontos-a-executar, a usar CPU e a realizar I/O).

- **tempo de espera** – qtd. tempo que um processo esteve à espera na fila dos prontos-a-executar.

- **tempo de resposta** – qtd. de tempo entre a submissão de um pedido e a primeira resposta produzida (não é output). Importante para time-sharing.

- **equitativo (justo)** – garantir que cada processo obtém a sua parte de CPU (não fica esquecido).

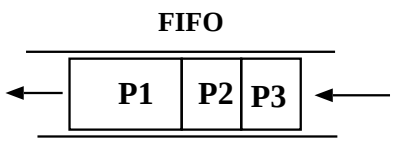
É desejável:

- Maximizar uso de CPU
- Maximizar throughput
- Minimizar tempo de turnaround
- Minimizar tempo de espera
- Minimizar tempo de resposta

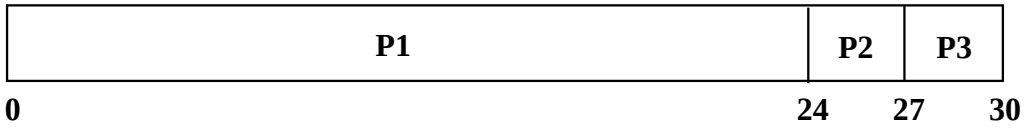
Algoritmos de Scheduling

→ **1º a chegar, 1º a ser seleccionado** (FCFS: First-Come, First-Served), é um algoritmo extremamente simples e fácil de implementar com uma fila (FIFO).

Processo	Tempo exec. (ms)
P1	24
P2	3
P3	3



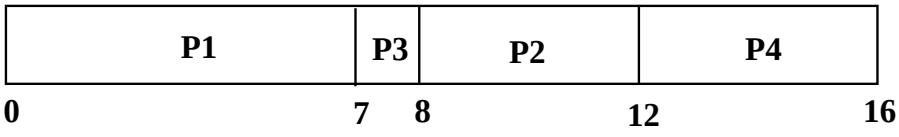
Obdecendo a ordem de chegada dos processos, o diagrama de Gantt para o escalonamento e':



- Tempo de espera para $P_1 = 0$; $P_2 = 24$; $P_3 = 27$.
- Tempo médio de espera: $(0 + 24 + 27)/3 = 17 milisecs$.

→ **Menor tarefa primeiro** (SJF: Shortest-job First): o CPU é atribuído ao processo que se pensa demorar menos tempo a executar.

Processo	Tempo chegada	Tempo exec. (ms)
P1	0.0	7
P2	0.2	4
P3	4.0	1
P4	5.0	4



- Tempo de espera para $P_1 = 0$; $P_2 = 8$; $P_3 = 7$; $P_4 = 12$.
- Tempo médio de espera: $(0 + 7 + 8 + 12)/4 = 6.75 milisecs$.
- Tempo médio de espera (FCFS): $(0+7+11+12)/4 = 7.5 milisecs$.

Menor tarefa primeiro: vantagens e inconvenientes

- é *óptimo* pois dá o menor tempo médio de espera para um dado conjunto de processos.
- *pouco praticável!*. Dificuldade em determinar a menor tarefa, pois não é possível saber-se antecipadamente qual o tempo que um dado processo precisa de CPU.
- permite *inanição* de processos (os que têm tempo de execução grande).
- Uma solução (com *preemption*– SRTF: Shortest Remaining Time First) para processos interactivos seria usar estimativas do tempo de execução e aplicar uma *técnica de envelhecimento* de modo a que os valores estimados à mais tempo tenham menor peso na nova estimativa. Considere-se:

  $$E_{n+1} = \alpha T_n + (1-\alpha)E_n = \\ \alpha T_n + (1-\alpha)\alpha T_{n-1} + (1-\alpha)^2 \alpha T_{n-2} + \ldots + (1-\alpha)^n E_1$$

  onde,

  T_i é o tempo de execução estimado para a execução do processo no instante i.
  E_i valor previsto para o instante i.
  E_1 valor previsto para o instante inicial.

  tomando $\alpha = 0.8$ teriamos:

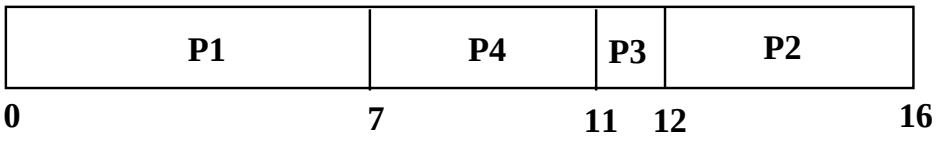
  $$E_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \ldots$$

  Quanto mais antiga for a observação menor peso na estimação actual.

Prioridades

- Neste algoritmo associa-se a cada processo um valor de prioridade. O processo com maior prioridade é escolhido pelo scheduler para aceder ao CPU.
- Valores menores de prioridade $\Rightarrow$ maior prioridade.
- Um exemplo para uma versão não-interruptível do algoritmo:

Processo	Tempo chegada	Tempo exec. (ms)	Prioridade
P1	0.0	7	3
P2	0.2	4	4
P3	4.0	1	3
P4	5.0	4	1



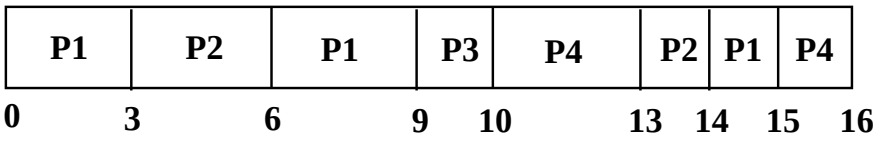
- Tempo médio de espera: $(0 + 7 + 11 + 12)/4 = 7.5 milisecs.$
- *Problema:* pode levar à inanição de processos. Um processo com prioridade baixa pode chegar a não ser escolhido para executar.
- *Solução:* técnica de envelhecimento, em que a prioridade de um processo aumenta com o tempo de espera, acabando por vir a ser seleccionado.

Round-Robin (distribuição circular de tempo)

- *time quantum* ou *time-slice* = intervalo mínimo de tempo de CPU atribuído a um processo (10-100 milisegundos).
- a cada processo é atribuído um quantum de tempo para executar. Decorrido esse tempo o processo é interrompido (*preempted*) e adicionado no fim da fila dos prontos-a-executar.
- quando um processo esgota o seu quantum, ou quando bloqueia ou termina a sua execução antes de esgotar o seu quantum, outro processo é escolhido para tomar o seu lugar, normalmente é o que está há mais tempo na fila.
- O quanto de tempo deve ser maior que o que se perde na troca de contexto entre os processos, senão não seria compensador!

Processo	Tempo chegada	Tempo exec. (ms)
P1	0.0	7
P2	0.2	4
P3	4.0	1
P4	5.0	4

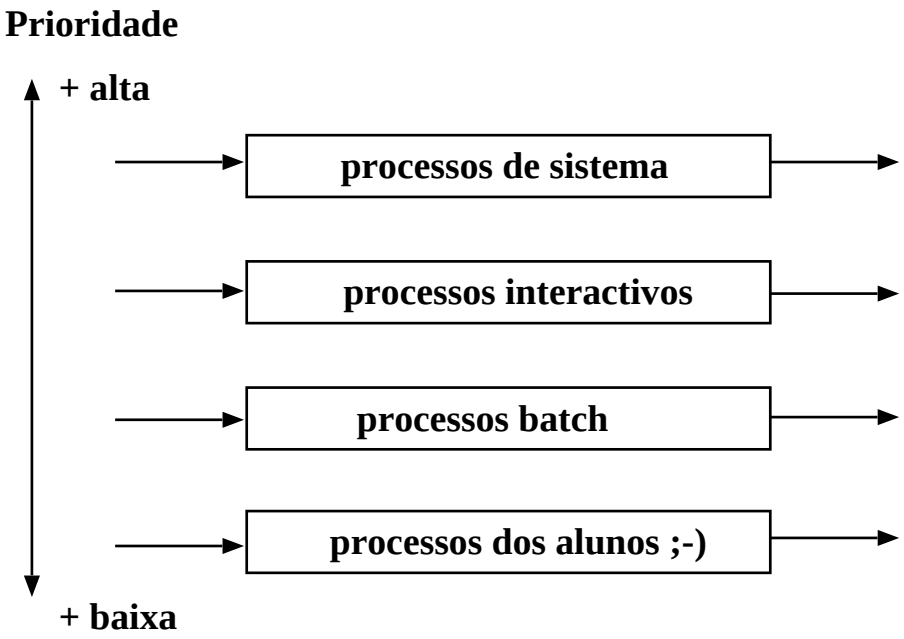
quantum=3



- Normalmente, tem um tempo médio de espera maior que SJF (SRTF), mas melhor tempo de resposta.

Filas de Níveis Múltiplos

- A fila dos prontos-a-executar é subdividida em várias filas, consoante os requisitos: *foreground* (interactivos) ou *background* (batch).
- cada fila tem o seu algoritmo de scheduling, e.g. interactivos é RR e os batch (FCFS).

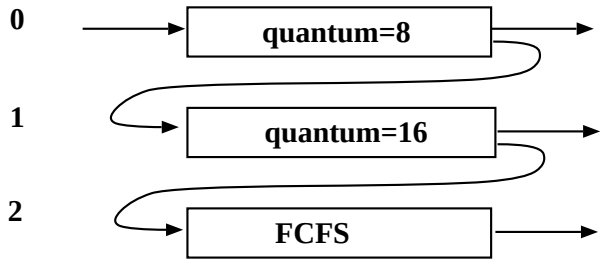


- um scheduler mais geral coordena os schedulers associados a cada fila, atribuindo uma fatia de tempo de CPU para execução de processos de uma dada fila, e.g.

  80% do tempo para processos interactivos,
  20% do tempo para processos batch.

Filas de Níveis Múltiplos com Realimentação

- Estende o algoritmo anterior, permitindo que os processos possam ser deslocados de umas filas para outras, consoante o uso que vão fazendo do CPU.
- Assim se um processo usar demasiado tempo de CPU, é deslocado para uma fila de mais baixa prioridade.
- Processos que estejam há muito tempo em filas de prioridade mais baixa vão sendo deslocados para filas de prioridade mais alta (*técnica do envelhecimento*), evitando-se inanição de processos.
- Parâmetros que determinam este scheduler:
  - número de filas
  - o algoritmo de scheduling de cada fila
  - método para promover um processo
  - método para despromover um processo
  - método para determinar qual a fila inicial onde o processo é colocado.
- Um exemplo:



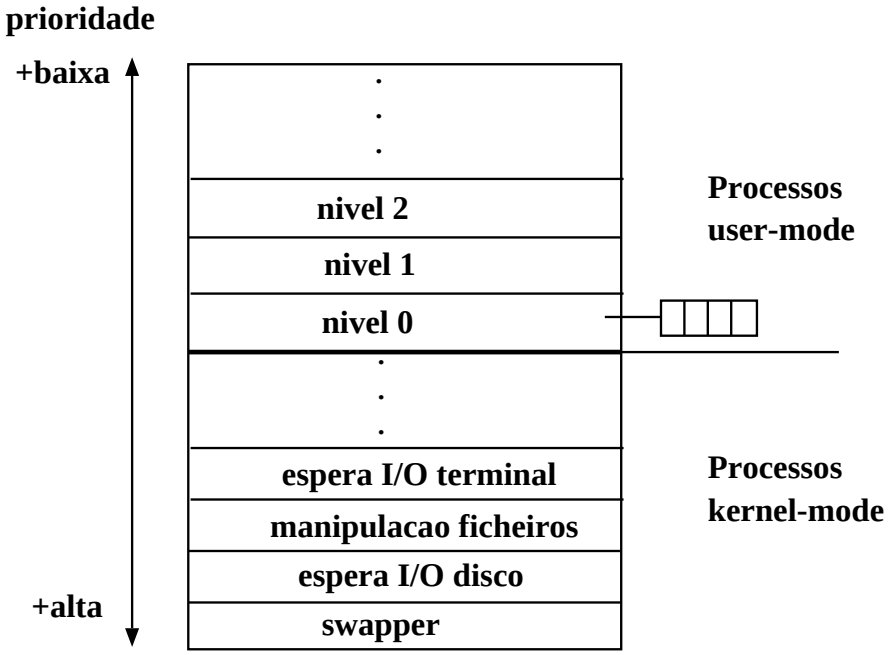
- Este é o algoritmo mais geral (e mais complexo) e pode fácilmente ser configurado para um SO em vista.

Scheduling (tradicional) em Unix

Unix é um sistema time-sharing, pelo que o algoritmo procura garantir que processos interactivos obtenham um bom tempo de resposta.

O algoritmo de scheduling tem dois níveis:

- *nível-baixo*: escolha do próximo processo a executar; usa filas múltiplas com um valor de prioridade associado a cada fila e um algoritmo Round-Robin dentro de cada fila.
- *nível-mais-alto*: partes dos processos são deslocados entre a memória e o disco, para que todos tenham a chance de estar em memória e serem executados.



- processos em execução em modo kernel têm prioridade mais alta (para que possam deixar o kernel o mais rápido possível!) do que em modo de utilizador.

Scheduling em Unix (cont.)

- as prioridades dos processos prontos-a-executar são actualizadas periodicamente, e.g. cada segundo, o que pode mudar o processo de nível.
- o objectivo é baixar a prioridade dos processos que mais usaram o CPU recentemente e fazer subir a prioridade dos processos que menos usaram o CPU.
- prioridade de um processo P_j varia com um instante i do seguinte modo:

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

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

onde,

$P_j(i)$	= prioridade do processo P_j no início do intervalo i;
$Base_j$	= prioridade de base do processo P_j;
$U_j(i)$	= tempo de CPU usado por P_j no intervalo i;
$CPU_j(i)$	= tempo médio de utilização do CPU por P_j até intervalo i;
$nice_j$	= valor de ajuste definido pelo utilizador;

- o contador de utilização de CPU para um dado processo é actualizado em cada tick do relógio, quando o processo está em execução.
- os processos interactivos têm normalmente um $U_j(i)$ baixo, porque suspendem muito, pelo que a sua prioridade é habitualmente alta.

Exemplo de actualização de prioridades