Sistemas Operativos: Implementação de Processos

Pedro F. Souto (pfs@fe.up.pt)

March 8, 2012

Sumário: Implementação de Processos

Contexto/Estado dum Processo (1/2)

- O SO tem que manter para cada processo alguma informação: o **contexto**/**estado** do processo
- Esta informação inclui:

  Informação genérica
  - O identificador do processo.
  - As *credenciais* do processo (incluindo o *owner*).

  Informação necessária para multiprocessamento
  - O estado do processo (READY,RUN,WAIT).
  - O evento pelo qual o processo está à espera, se algum.
  - O estado do CPU (PC, SP e outros registos), quando o SO o retirou do processo, i.e. saiu do estado *running*.

  Informação relativa a diferentes serviços do SO
  - Informação sobre a memória usada pelo processo.
  - Informação sobre outros recursos usados (p.ex. ficheiros) pelo processo e o estado desses recursos.

Contexto/Estado dum Processo (2/2)

SO monolíticos:

- Esta informação é guardada numa estrutura de dados conhecida por *process control block (PCB)*.
- Os PCBs de todos os processos são mantidos na *process table*.

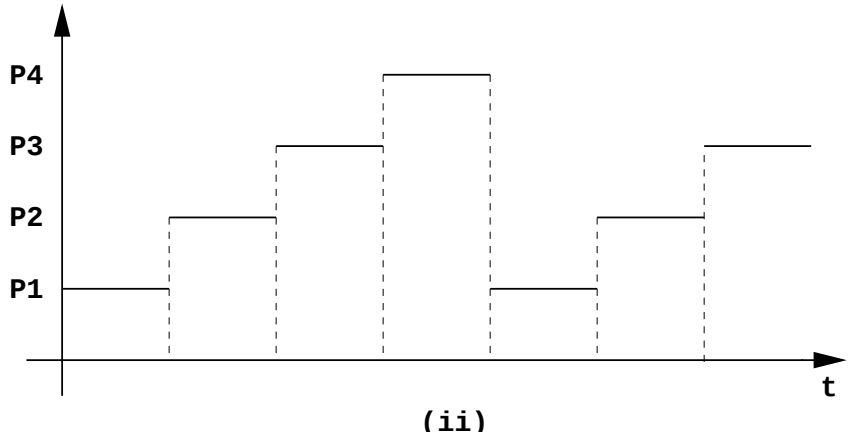
SO baseados em *micro-kernel* esta informação está dispersa:

- Pelo *micro-kernel*
- Os diferentes processos de sistema, cada um dos quais mantém esta informação uma *process table* privada.

Sumário: Implementação de Processos

Comutação de Processos (1/2)

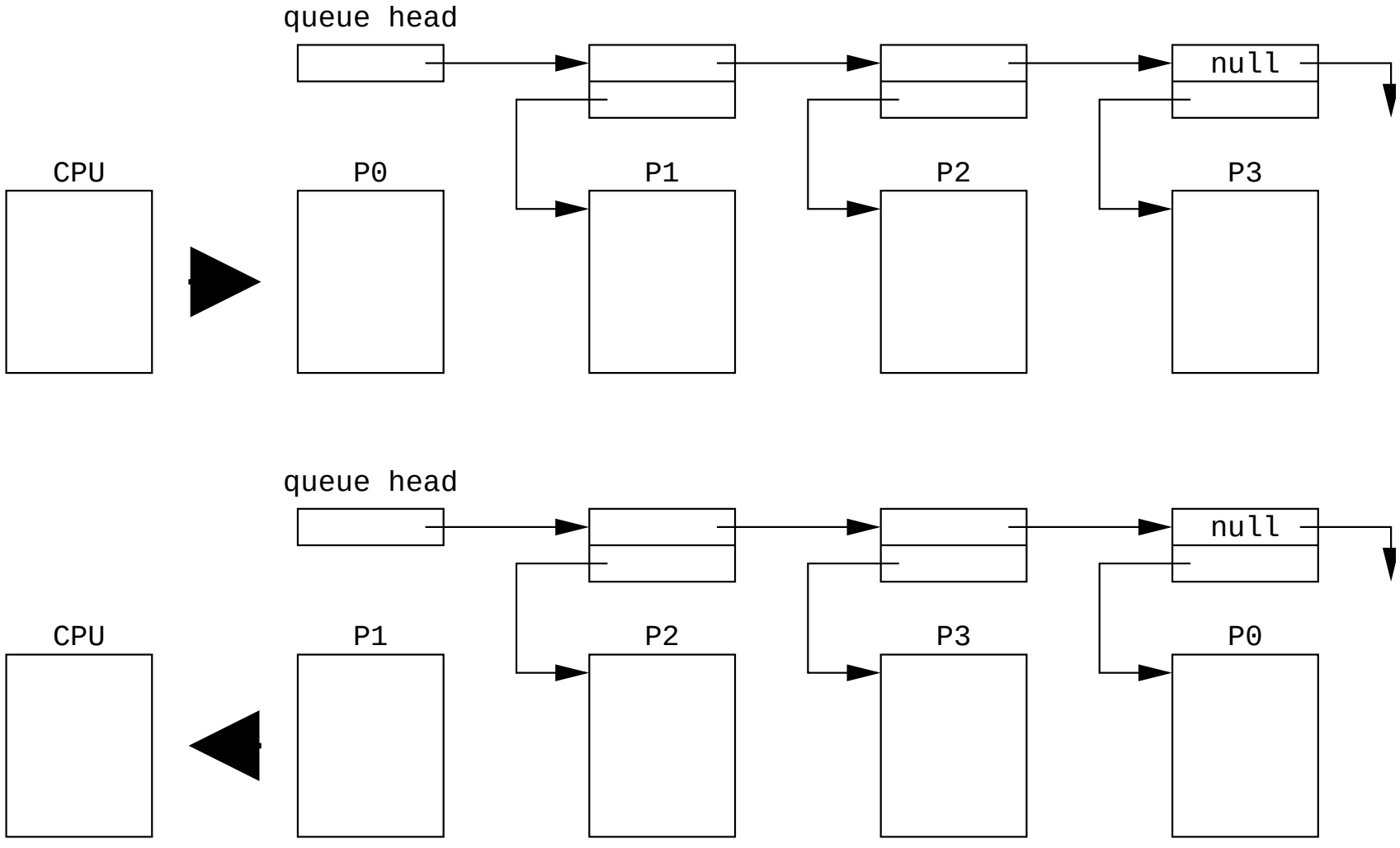
- A comutação entre processos (*process switching*), consiste em *retirar* o CPU a um processo e *atribuí-lo* a outro.
  - É uma função fundamental do *kernel* em SO multiprocesso



- A **comutação entre processos** inclui:
  1. Transferir o estado dos registos do processador para o "PCB" do processo em execução;
  2. Carregar os registos do processador com os valores previamente guardados no "PCB" do processo a executar;
     - Passar o controlo para o novo processo, possivelmente comutando para *user-mode*.

Comutação de Processos (2/2)

- P_0 está no estado *RUN* e passa para o estado *READY*:



Despacho

- A parte do SO que faz a comutação de processos designa-se por despacho *(dispatcher)*.
- A comutação entre processos é feita em instantes "oportunos", incluindo:
  - chamadas ao sistema (p.ex., `exit()`, `read()`);
  - interrupções:
    - por dispositivos de E/S;
    - pelo relógio.
- Frequentemente a implementação de chamadas ao sistema usa o mesmo mecanismo que interrupções, p.ex. a instrução `INT`
  - A implementação da comutação de processos é semelhante em ambos os casos

Processamento de Interrupções sem SO

(Estudado em Computadores/Sistemas Baseados em Microprocessadores)

1. O *hardware* salva o estado do CPU na *stack*
2. O *hardware* carrega o PC com o vector de interrupção
   - Assumindo que o processador suporta interrupções vectorizadas
3. Uma rotina em *assembly* guarda registos que o *hardware* não tenha salvado
4. A interrupção é servida por uma rotina possivelmente em C
5. Outra rotina em *assembly*:
   - restaura os registos salvados em 3;
   - executa `RETI`

A execução do "processo interrompido" é retomada.

Comutação de Processos com Interrupções

Ideia Em vez de retomar a execução do processo interrompido, pode retomar-se a execução de outro processo no estado `RDY`

Implementação

1. Guardar o contexto do processo interrompido antes de executar o *interrupt handler*
2. Seleccionar um processo para executar e repor o seu contexto após executar o *interrupt handler*

Outras diferenças

- Execução em *kernel mode* usa uma *stack* diferente da dos processos interrompidos
  - Garante que a *stack* tem espaço suficiente

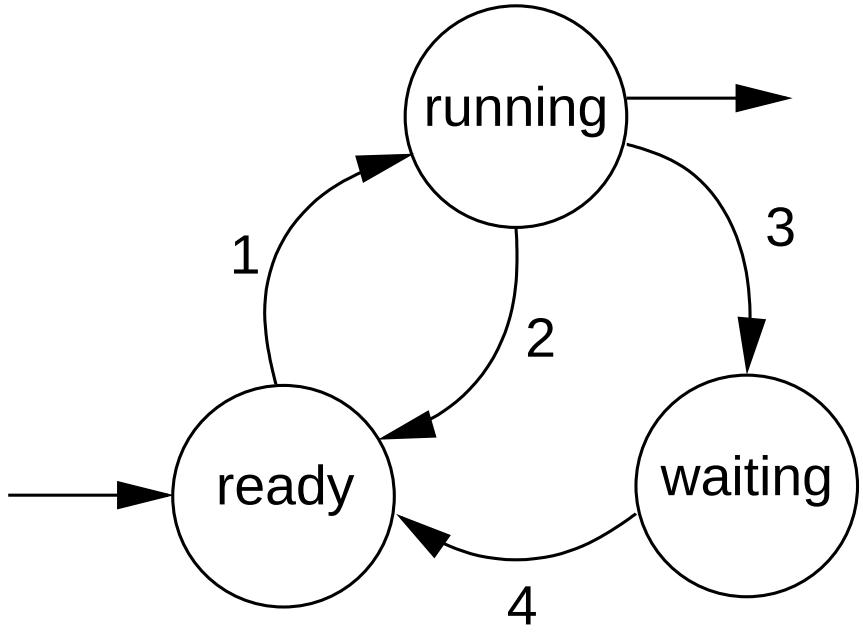
Eficiência

- O desempenho do despacho é crítico:
  - É executado frequentes vezes
  - É puro *overhead*
- Depende não só do SO mas também do HW

Sumário: Implementação de Processos

Escalonamento de Processos

Problema: quando há mais de um processo em condições de execução (estado *ready*), qual o processo a executar?



Solução: o SO, mais precisamente o *scheduler*, executa um algoritmo de escalonamento para decidir.
I.e., o escalonador determina a que processo deve ser atribuído o CPU.

Critérios para um Bom Algoritmo de Escalonamento

- **Equidade**: dar oportunidade a todos os processos para progredir.
- **Equilíbrio**: na utilização dos recursos.
- Satisfação da política escolhida:
  - **Sistemas interactivos**:
    - tempo de resposta;
    - previsibilidade.
  - **Sistemas de tempo-real**:
    - cumprir prazos;
    - determinismo.
  - **Sistemas não-interactivos**:
    - débito;
    - utilização do CPU.

Escalonamento *Preemptivo vs. Não-preemptivo*

- Um algoritmo de escalonamento diz-se **não-preemptivo**, se, uma vez na posse do CPU, um processo executa até o (ao CPU) "libertar" voluntariamente.
- Quase todos os algoritmos de escalonamento têm duas versões: uma preemptiva outra não.
- Algoritmos não-preemptivos têm problemas graves:
  - certas classes de processos executam durante muito tempo até bloquear;
  - um utilizador *egoísta* pode impedir que o computador execute processos de outros utilizadores.
- Praticamente todos os sistemas operativos usam algoritmos preemptivos:

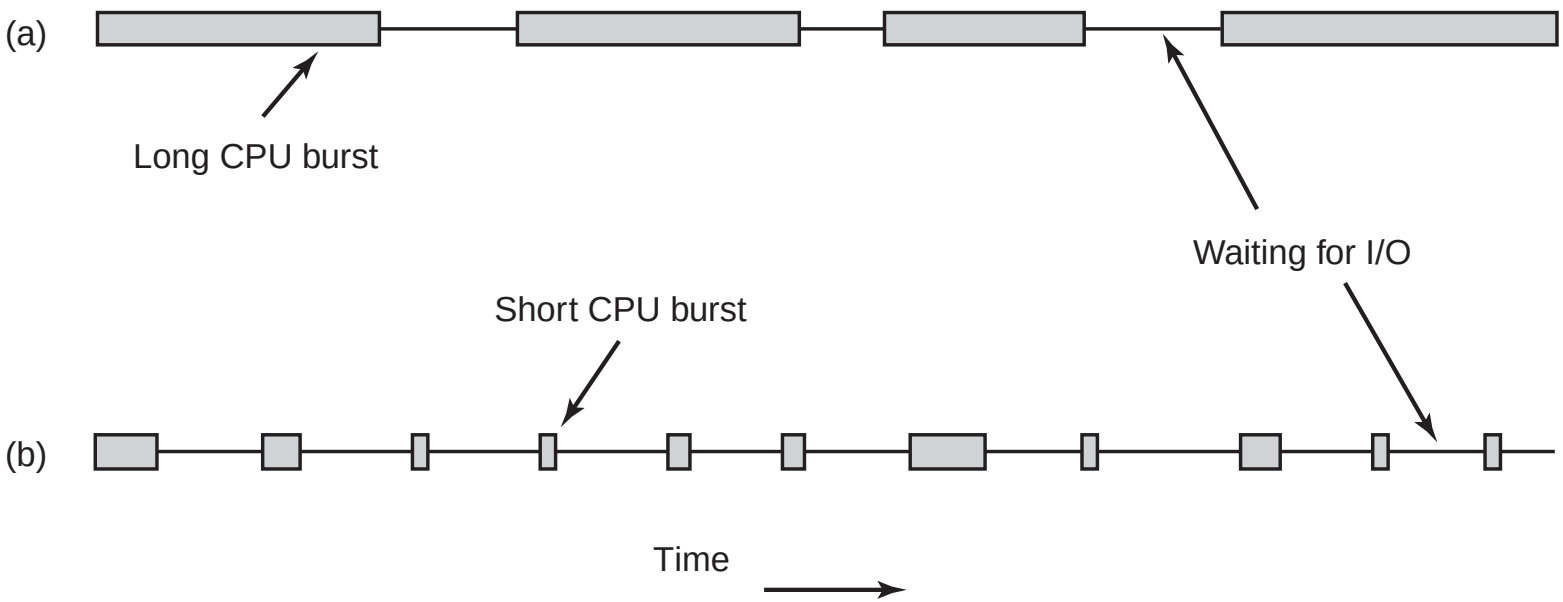
  O SO usa as interrupções do relógio para retirar o CPU ao processo em execução.

Algoritmo *Round-Robin*

- Quando um processo passa para o estado de **execução**, o SO atribui-lhe a posse do CPU por um intervalo de tempo fixo: quantum.
- A comutação de processos realiza-se:
  - ou quando o processo em execução bloquear/terminar;
  - ou quando o *quantum* expirar.
- Qual o valor dum *quantum*?
  - compromisso tempo-de-resposta/utilização-do-CPU: a comutação de processos leva algum tempo;
  - Unix/Linux usa um valor de 100 ms, que se mantém inalterado há mais de 20 anos!!!
- Algumas vantagens deste algoritmo são:
  - fácil de implementar (como?);
  - equitável (mais uma característica).

Padrão de Execução dum Processo

- Processos (e *threads*) alternam:
  - execução de instruções;
  - realização de operações de entrada/saída.
- De acordo com a duração dos intervalos de execução de instruções, um processo pode ser classificado em *CPU-bound* (a) ou *IO-bound* (b)



Escalonamento Baseado em Prioridades

Problema por vezes, alguns processos são mais iguais do que os outros.

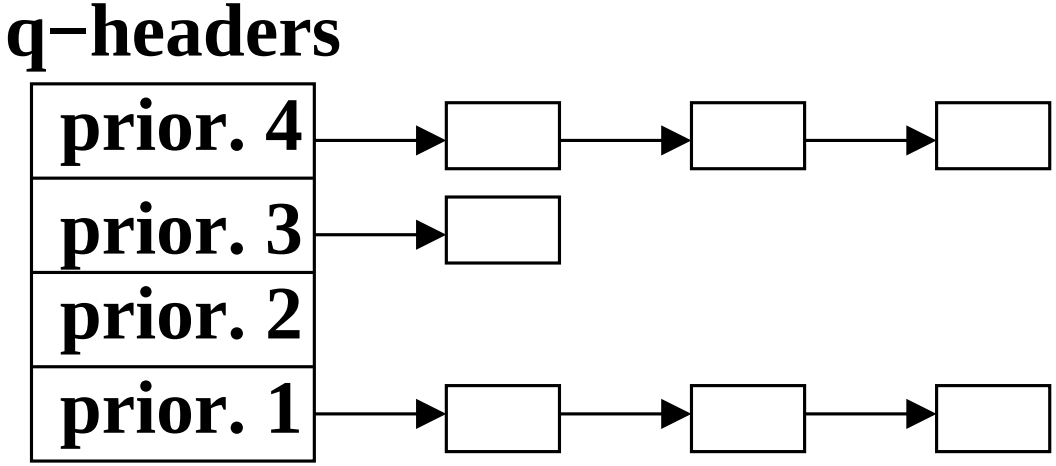
Solução atribuir uma prioridade a cada processo. Quando da comutação de processos, o processo pronto a executar com maior prioridade passa para o estado de **execução**.

- As prioridades podem ser atribuídas estática ou dinamicamente.
- Potencial problema: míngua (*starvation*):

  Os processos de mais baixa prioridade podem nunca ser seleccionados para executar.

Escalonamento em BSD-Unix

- Cada processo tem uma prioridade.
- Processos são agrupados em classes com base na sua prioridade:
  - o escalonamento entre classes é baseado em prioridades;
  - o escalonamento numa classe usa *round-robin*.



- Para evitar míngua de processos de prioridade mais baixa:

  sempre que um processo executa até ao fim do seu quantum, a sua prioridade diminui.

Rate-monotonic Scheduling

- Assume n processos periódicos, cada um deles:
  - de período T_i igual ao prazo (*deadline*);
  - com um tempo de execução C_i.
- É de facto um algoritmo de atribuição de prioridades estáticas:

  quanto mais curto é período maior é a prioridade.

- Tem propriedades matemáticas que o tornam apropriado para sistemas *hard real-time*.
  - Neste tipo de sistemas é fundamental ser capaz de determinar a satisfação ou não dos requisitos temporais do conjunto de processos
- O algoritmo original (1973) impõe sérias restrições, mas desenvolveram-se técnicas que permitem levantar a maioria delas.

Mecanismos *vs.* Políticas

Observação: Não é possível “agradar a gregos e a troianos.”:

- O SO não conhece as necessidades das aplicações suficientemente para tomar as decisões mais apropriadas.

Ideia: oferecer funcionalidade básica e permitir que os seus utilizadores a usem da maneira mais conveniente

- O SO oferece os mecanismos:
  - algoritmos de escalonamento parametrizáveis.
- Os processos seleccionam as políticas:
  - i.e. o algoritmo e os valores dos seus parâmetros.

Escalonamento em POSIX (Linux) (1/2)

- POSIX define 3 políticas de escalonamento:

  `SCHED_FIFO` preemptivo, baseado em prioridades estáticas;

  `SCHED_RR` semelhante a `SCHED_FIFO` mas com *quanta*;

  `SCHED_OTHER` algoritmo não especificado, i.e. dependente da implementação.

- O escalonador é baseado em prioridades.
- A política de escalonamento especifica:
  - quando os processos passam do estado *run* para o estado *ready*;
  - onde é que são inseridos na *ready-list* de processos com a mesma prioridade.

Escalonamento em POSIX (Linux) (2/2))

- `SCHED_FIFO` e `SCHED_RR` foram especificados para aplicações de tempo-real. A sua prioridade é sempre superior à de processos com política `SCHED_OTHER`.
- Em Linux, `SCHED_OTHER` é um algoritmo semelhante ao de BSD-Unix, embora os pormenores sejam ligeiramente diferentes. (Para mais informação veja `man 2 sched_setscheduler`.)
- Estas políticas de escalonamento podem também ser atribuídas a *threads* através de funções de *libpthread*:

```
int pthread_setschedparam(pthread_t tid,
        int policy, struct sched_param *param);
```

Sumário: Implementação de Processos

Leitura Adicional

Sistemas Operativos

- Secções 4.1, 4.2, 4.3 e 4.4

Modern Operating Systems, 2nd. Ed.

- Secção 2.1.6: *Implementation of Processes*
- Secção 2.5: *Scheduling*
- Secção 10.3: *Processes in Unix*

Operating Systems Concepts

- Secções 3.1.3, 3.2: *Process Control Block* e *Scheduling*
- Secções 5.1, 5.2 , 5.3 (e 5.4): *Scheduling*