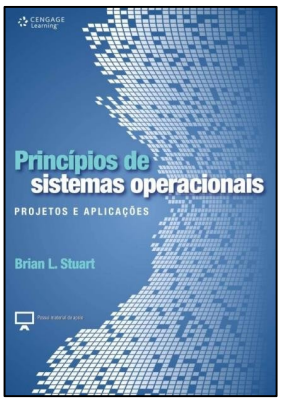
FACULDADE DE COMPUTAÇÃO E INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
SISTEMAS OPERACIONAIS – Aula 07 – 2º SEMESTRE/2019
PROF. LUCIANO SILVA

TEORIA: GERENCIADOR DE MEMÓRIA (PARTE I)

Nossos **objetivos** nesta aula são:

- conhecer o mecanismo de criação de jobs e sua transformação em processos
- conhecer o mecanismo de *swapping* de processos



Para esta aula, usamos como referência as **Seções 9.1 a 9.3 do Capítulo 9 (Gerenciamento de Memória)** do nosso livro-texto:

STUART, B.L., **Princípios de Sistemas Operacionais: Projetos e Aplicações**. São Paulo: Cengage Learning, 2011.

Não deixem de ler estas seções depois desta aula!

CRIAÇÃO DE JOBS e TRANSFORMAÇÃO EM PROCESSOS

- Sabemos, das aulas anteriores, que um **processo** é originário de um **job**. Um **job** é como o **gerenciador de memória "enxerga"** o seu programa em execução na memória que esteja gerenciando. Já um **processo** é como o **gerenciador de processos "enxerga"** o seu programa em execução. No MINIX 2.0, há duas estruturas de dados para cada um deste conceitos:

Job em MINIX 2.0 – mproc.h	Processo em MINIX 2.0 – proc.h

struct mproc {
struct mem_map mp_seg[NR_SEGS];
…
pid_t mp_pid;
…
}
	struct proc{
	struct mem_map p_map[NR_SEGS];
	…
	pid_t p_pid;
	…
	}

- Vimos que, ambas as estruturas, armazenam o mesmo mapa de memória do programa em execução, embora com nomes diferentes. A transformação de um job em um processo, normalmente, é feita utilizando o mecanismo de **fork**. Uma vez que o job esteja carregado em memória (com as estratégias que veremos a seguir), usamos um fork para criar a primeira cópia do job em processo. Abaixo, temos a função fork implementada em MINIX 2.0:

```
PUBLIC int do_fork()
{
/* The process pointed to by 'mp' has forked.  Create a child process. */


  register struct mproc *rmp;/* pointer to parent */
  register struct mproc *rmc;/* pointer to child */
  int i, child_nr, t;
  phys_clicks prog_clicks, child_base;
  phys_bytes prog_bytes, parent_abs, child_abs;    /* Intel only */


 /* If tables might fill up during FORK, don't even start since recovery half
  * way through is such a nuisance.
  */
  rmp = mp;
  if (procs_in_use == NR_PROCS) return(EAGAIN);
  if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0)return(EAGAIN);


  /* Determine how much memory to allocate.  Only the data and stack need to
   * be copied, because the text segment is either shared or of zero length.
   */
  prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
  prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
  if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);


  /* Create a copy of the parent's core image for the child. */
  child_abs = (phys_bytes) child_base << CLICK_SHIFT;
  parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
  i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
  if (i < 0) panic("do_fork can't copy", i);
```

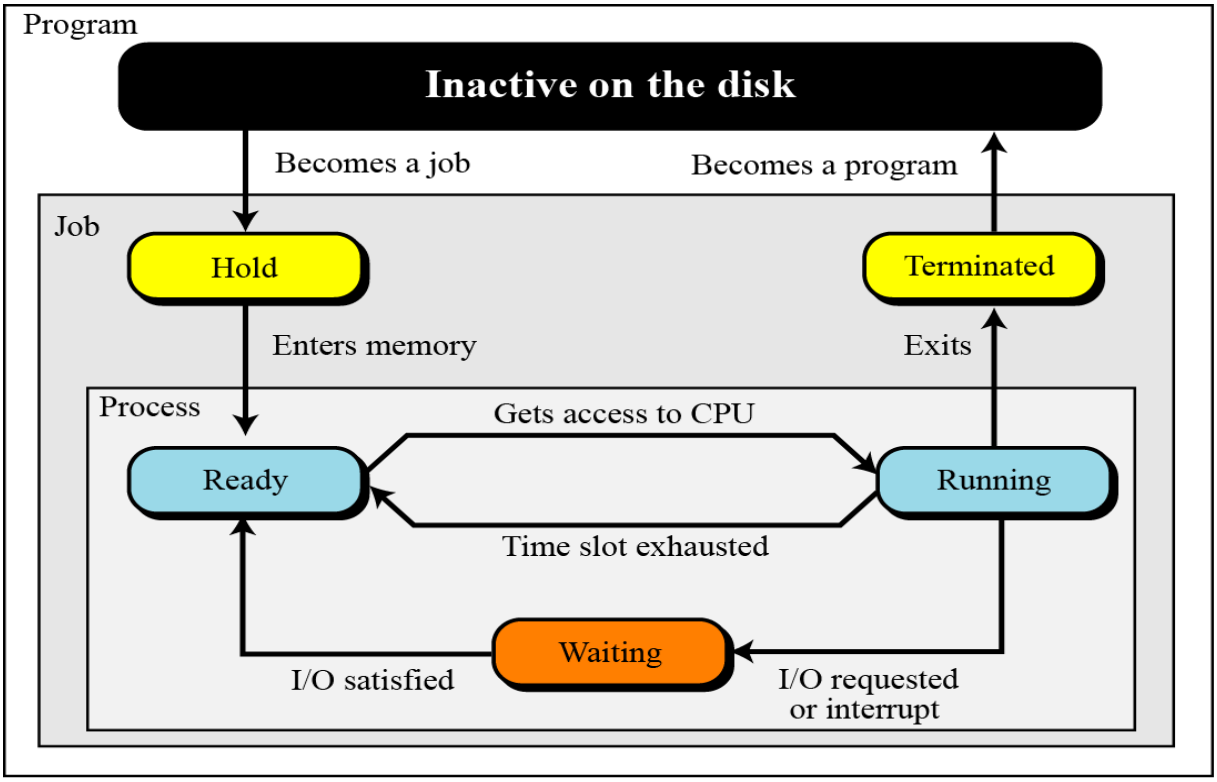
```
/* Find a slot in 'mproc' for the child process.  A slot must exist. */
for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
      if ( (rmc->mp_flags & IN_USE) == 0) break;


/* Set up the child and its memory map; copy its 'mproc' slot from parent. */
child_nr = (int)(rmc - mproc);     /* slot number of the child */
procs_in_use++;
*rmc = *rmp;               /* copy parent's process slot to child's */


rmc->mp_parent = who;                     /* record child's parent */
rmc->mp_flags &= (IN_USE|SEPARATE);       /* inherit only these flags */


/* A separate I&D child keeps the parents text segment.  The data and stack
 * segments must refer to the new copy.
 */
if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
rmc->mp_seg[D].mem_phys = child_base;
rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
                    (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
rmc->mp_exitstatus = 0;
rmc->mp_sigstatus = 0;


/* Find a free pid for the child and put it in the table. */
do {
      t = 0;                /* 't' = 0 means pid still free */
      next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
      for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
            if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
                  t = 1;
                  break;
            }
      rmc->mp_pid = next_pid;       /* assign pid to child */
} while (t);


/* Tell kernel and file system about the (now successful) FORK. */
sys_fork(who, child_nr, rmc->mp_pid);
tell_fs(FORK, who, child_nr, rmc->mp_pid);


/* Report child's memory map to kernel. */
sys_newmap(child_nr, rmc->mp_seg);
```

```
  /* Reply to child to wake it up. */
  setreply(child_nr, 0);
  return(next_pid);           /* child's pid */
}
```

- Devido a este mecanismo de criação de processos a partir de jobs, em que um processo pode ser considerado "um filho" do job, alguns sistemas operacionais podem adotar definições diferentes para jobs e processos. Em **Linux**, por exemplo, **um job é um considerado um grupo de processos**. Em Linux, quando um **job tem um único processo**, ele é chamado de **task (ou tarefa)**. Um processo, em Linux, segue o conceito tradicional de processo.

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

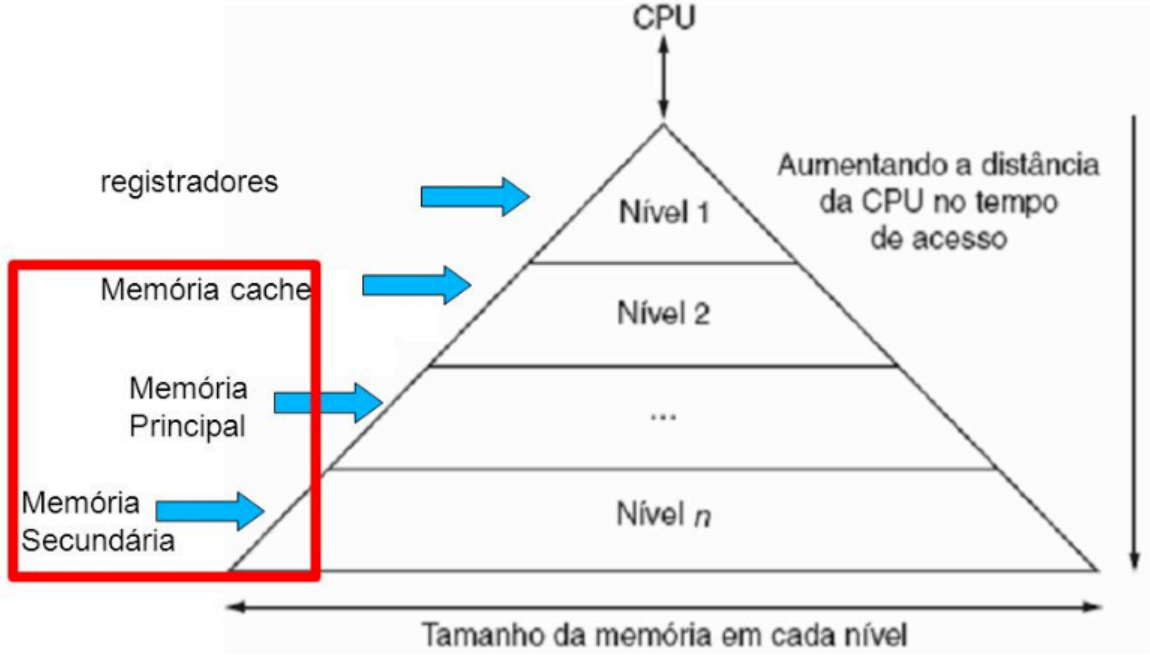
Vamos reconsiderar o diagrama visto nas Aula 02, onde relacionamos o gerenciados de jobs e o gerenciador de processos.



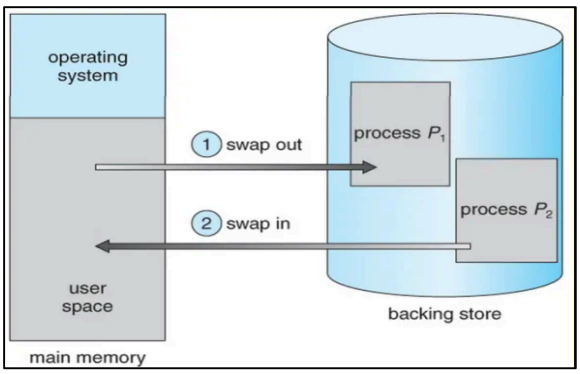
Sabemos que o gerenciador de processos controla os três estados básicos (pronto, execução e espera). Quais os estados controlados pelo gerenciador de jobs ? Ele necessita de uma fila de job, assim como o gerenciador de processos ?

MEMÓRIA VIRTUAL

- Abaixo, recordamos a hierarquia de memória, com destaque para as memórias primária e secundária.



- Nos esquemas de memória virtual, o SO entende a memória secundária (discos, fitas, etc) como uma extensão da memória primária (RAM/ROM). Ele mantém na memória principal o que será necessário, espacialmente ou temporalmente, para realizar o processo. O processo que, no momento, não esteja sendo utilizado pela memória principal, pode ficar armazenado temporariamente numa **área com formatação especial** chamada **swap**. O termo **swapping** refere-se ao mecanismo de troca de regiões entre a memória principal e a memória secundária. Existe outra técnica de memória virtual, chamada paginação, que veremos na próxima aula.



EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Atualmente, um sistema operacional consegue executar um programa cujo tamanho seja melhor que a memória principal disponível ? Justifique.

EXERCÍCIOS EXTRA-CLASSE

1. Pesquise três sistemas operacionais diferentes e mostre como eles definem job, task e processo.

2. O esquema de criação de processos visto em aula (programa→ job → processo) não é um mecanismo muito rápido para criação de processos. Identifique, utilizando a implementação do MINIX 2.0, duas formas de otimizar a criação de processos.

3. Sabemos que alguns processadores que estão sob o controle de um SO possuem memórias cache internas. Mostre duas maneiras diferentes que um SO poderia utilizar estas memórias cache para otimizar a execução de programas.

4. No esquema de memória virtual com swapping, levamos o processo inteiro para a região de swap e vice-versa. Seria realmente necessário levar o processo inteiro ? Justifique.