MAC422
Sistemas Operacionais

Prof. Marcel P. Jackowski

Aula #4

Criação de processos e mecanismos de comunicação

Referências: Sistemas Operacionais Modernos (3a. Ed.), aulas antigas de MAC-422 (IME-USP), W4118 (Columbia University)

Criação de processos (i)

- Em Unix, Linux e variações:
  - Utiliza-se a função fork() para criar um novo processo e exec() para executar um programa externo.
- Por que não uma única chamada fork_and_exec() ?
  - Pense como você implementaria "ls | wc" ?

Criação de processos (ii)

- Option 1: cloning (e.g., Unix fork(), exec())
  - Pause current process and save its state
  - Copy its PCB (can select what to copy)
  - Add new PCB to ready queue
  - Must distinguish parent and child
- Option 2: from scratch (Win32 CreateProcess)
  - Load code and data into memory
  - Create and initialize PCB (make it like saved from context switch)
  - Add new PCB to ready queue

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int a = 2;
  int b = -1;
  int c = 3;

  fork();

  a = a + 2;
  printf("pid=%d, a=%d, b=%d, c=%d\n",getpid(),a,b,c);

  fork();

  b = a + c;
  printf("pid=%d, a=%d, b=%d, c=%d\n",getpid(),a,b,c);

  fork();

  c = a + b;
  printf("pid=%d, a=%d, b=%d, c=%d\n",getpid(),a,b,c);

  exit(0);
}
```

- Quantos processos serão criados ?
- Quantas vezes serão impressos os valores de a, b e c ?
- Eles serão diferentes entre execuções ?

Término de processos

- Normal: exit(int status)
  - OS passes exit status to parent via wait(int *status)
  - OS frees process resources
- Abnormal: kill(pid_t pid, int sig)
  - OS can kill process
  - Process can kill process

Entre zumbis e órfãos

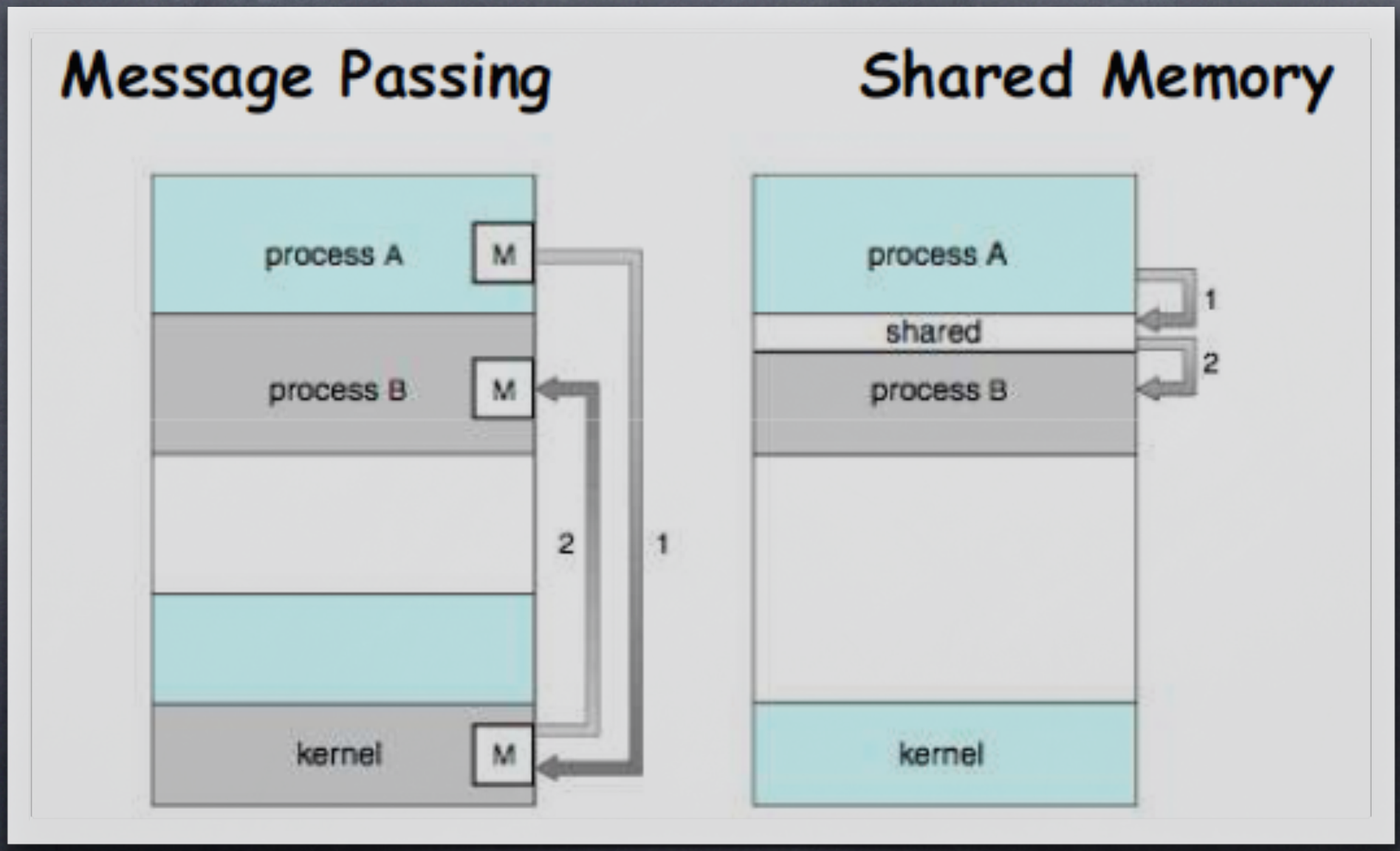
- What if child exits before parent?
  - Child becomes zombie
    - Need to store exit status
    - OS can't fully free
  - Parent must call wait() to reap child

- What if parent exits before child?
  - Child becomes orphan
    - Need some process to query exit status and maintain process tree
  - Re-parent to the first process, the init process

Processos cooperativos

- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity/Convenience

Comunicação interprocessos (IPC)



Troca de mensagens

Memória compartilhada

Vantagens e desvantagens

- Message passing
  - Why good?
  - Why bad?
- Shared Memory
  - Why good?
  - Why bad?

Exemplo de IPC: Sinais em Unix/Linux

- Signals
  - A very short message: just a small integer
  - A fixed set of available signals. Examples:
    - 9: kill
    - 11: segmentation fault
- Installing a handler for a signal
  - sighandler_t signal(int signum, sighandler_t handler);
- Send a signal to a process
  - kill(pid_t pid, int sig)

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigproc()
{ /* NOTE some versions of UNIX will reset signal to default
     after each call. So for portability reset signal each time
     signal(SIGINT, sigproc);
  */
  printf("you have pressed ctrl-c...\n");
}

void sigkill()
{
  printf("a-ha! you can't kill me!\n");
}

void sigusr1()
{
  printf("Hi, what do you want?\n");
}

int main()
{
  signal(SIGINT, sigproc);
  signal(SIGKILL, sigkill);
  signal(SIGUSR1, sigusr1);
  for(;;); /* infinite loop */
}
```

Exemplo de IPC: Pipes

- int pipe(int fds[2])
  - Creates a one way communication channel
  - fds[2] is used to return two file descriptors
  - Bytes written to fds[1] will be read from fds[0]

```
int pipefd[2];
pipe(pipefd);
        switch(pid=fork()) {
        case -1: perror("fork"); exit(1);
        case 0:  close(pipefd[0]);
                  // write to fd 1
                  break;
        default:  close(pipefd[1]);
                  // read from fd 0
           break;
}
```

Exemplo de IPC: Pipes (ii)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];

    pipe(pfds);

    if (!fork()) {
        close(1);       /* close normal stdout */
        dup(pfds[1]);   /* make stdout same as pfds[1] */
        close(pfds[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* close normal stdin */
        dup(pfds[0]);   /* make stdin same as pfds[0] */
        close(pfds[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

ls | wc

Exemplo de IPC: Memória compartilhada

- int shmget(key_t key, size_t size, int shmflg);
  - Create a shared memory segment
  - key: unique identifier of a shared memory segment, or IPC_PRIVATE

- int shmat(int shmid, const void *addr, int flg)
  - Attach shared memory segment to address space of the calling process
  - shmid: id returned by shmget()

- int shmdt(const void *shmaddr);
  - Detach from shared memory

- Problem: synchronization!

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
```

```
    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }

    /* attach to the segment to get a pointer to it: */
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```

Informações sobre processos

- ps
- top
- For each process, there is a corresponding directory /proc/<pid> to store this process information in the /proc pseudo file system

ls /proc

mjack@ubuntu: ~/MAC0422-2011/xv6

File Edit View Terminal Help

```
mjack@ubuntu:~/MAC0422-2011/xv6$ ls /proc
1       1376  1602  1683   232   3907  690      buddyinfo    key-users      self
10      14    1606  1685   24    4     695      bus          kmsg           slabinfo
1006    15    1619  1687   249   4004  696      cgroups      kpagecount     softirqs
1016    1506  1624  1695   250   4005  7        cmdline      kpageflags     stat
1030    1510  1625  17     27    4006  762      cpuinfo      latency_stats  swaps
1039    1528  1627  1703   28    41    788      crypto       loadavg        sys
10856   1562  1629  1705   29    42    8        devices      locks          sysrq-trigger
1091    1565  1631  1707   3     43    852      diskstats    mdstat         sysvipc
10927   1566  1633  1716   30    44    856      dma          meminfo        timer_list
1097    1569  1644  18     309   45    862      driver       misc           timer_stats
11      1576  1645  19     31    46    863      execdomains  modules        tty
1120    1578  1655  2      312   5     865      fb           mounts         uptime
12      1586  1658  20     32    518   866      filesystems  mpt            version
1240    1589  1659  21     3460  6     870      fs           mtrr           version_signature
1278    1591  1660  22     3461  625   871      interrupts   net            vmallocinfo
1282    1592  1671  22035  35    653   889      iomem        pagetypeinfo   vmmemctl
1286    1594  1674  2240   3518  673   9        ioports      partitions     vmstat
1292    1596  1676  23     37    679   935      irq          sched_debug    zoneinfo
13      1598  1679  230    3771  686   acpi     kallsyms     schedstat
1340    16    1680  231    38    688   asound   kcore        scsi
mjack@ubuntu:~/MAC0422-2011/xv6$
```

Processo vs. “thread”

- Thread: separate streams of execution that share the same address space

- Process != Thread
  - One process can have multiple threads
  - Threads communicate more efficiently

“thread” : fio

Linux: tasks!

- Linux uses a neutral term: tasks
  - Tasks represent both processes and threads
  - When processes trap into the kernel, they share the Linux kernel's address space → kernel threads

Um processo possui pelo menos 1 thread!

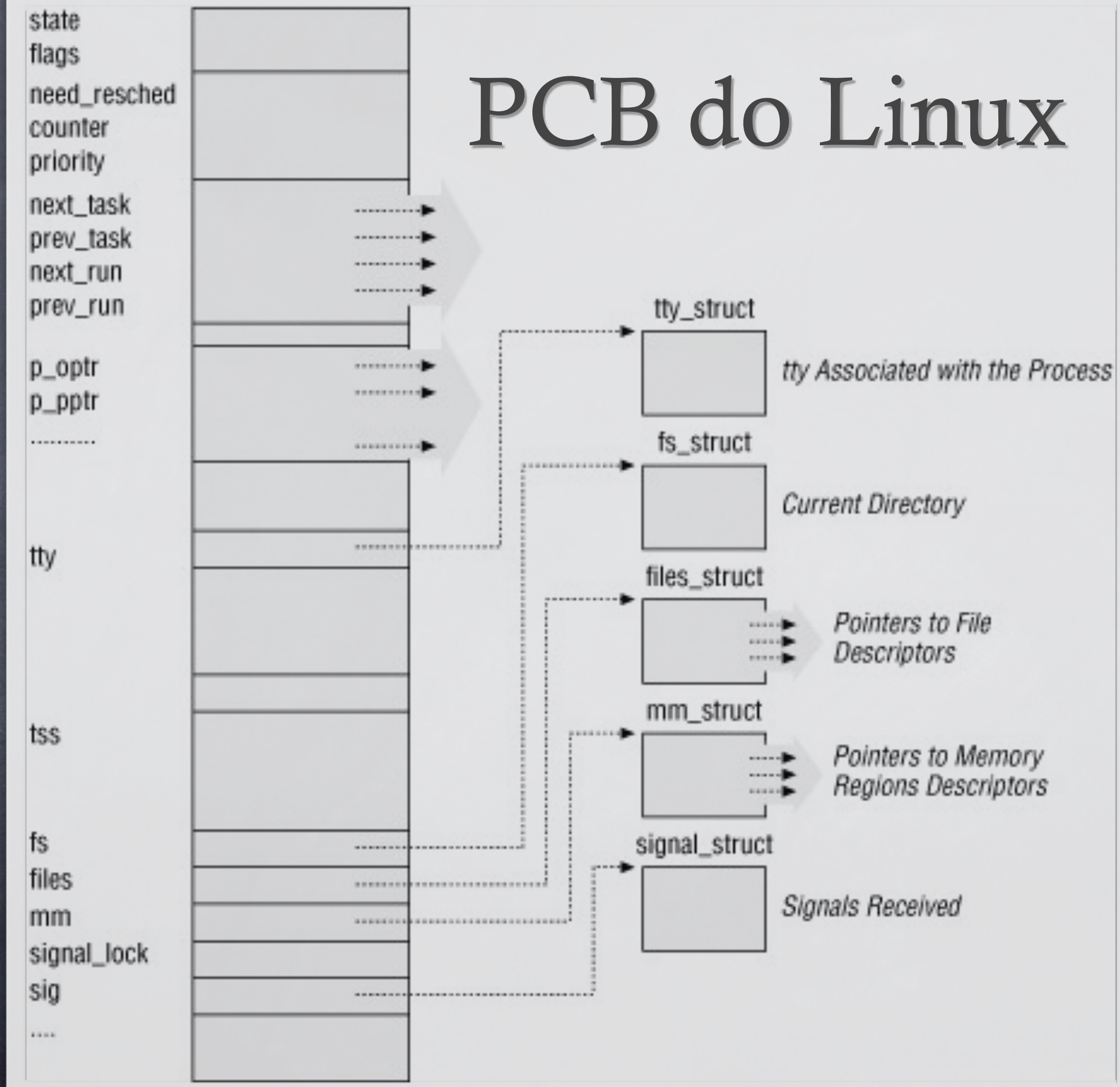
Como criar threads ?

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* thread_fn(void *arg)
{
    int id = (int)arg;
    printf("thread %d runs\n", id);
    return NULL;
}

int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_fn, (void*)1);
    pthread_create(&t2, NULL, thread_fn, (void*)2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

PCB do Linux


state
flags
need_resched
counter
priority
next_task
prev_task
next_run
prev_run
p_optr
p_pptr
.....
tty
tss
fs
files
mm
signal_lock
sig
....
tty_struct
tty Associated with the Process
fs_struct
Current Directory
files_struct
Pointers to File Descriptors
mm_struct
Pointers to Memory Regions Descriptors
signal_struct
Signals Received

Estados de um processo em Linux

- state: what state a process is in
  - TASK_RUNNING – the thread is running on the CPU or is waiting to run
  - TASK_INTERRUPTIBLE – the thread is sleeping and can be awoken by a signal (EINTR)
  - TASK_UNINTERRUPTIBLE – the thread is sleeping and cannot be awakened by a signal
  - TASK_STOPPED – the process has been stopped by a signal or by a debugger
  - TASK_TRACED – the process is being traced via the ptrace system call
- exit_state: how a process exited
  - EXIT_ZOMBIE – the process is exiting but has not yet been waited for by its parent
  - EXIT_DEAD – the process has exited and has been waited for

Identificadores de processos

- process ID: pid
- thread group ID: tgid
  - pid of first thread in process
  - getpid() returns this ID, so all threads in a process share the same process ID
- many system calls identify a process by its PID
  - Linux kernel uses pidhash to efficiently find processes by pids
  - see include/linux/pid.h, kernel/pid.c

getpid()

- Processos do usuário não podem executar operações privilegiadas por si próprios
  - Exemplo: getpid()
- Processos devem pedir ao SO para realizar tais chamadas através de "system calls"
- O SO é responsável por validar os parâmetros de cada um destes pedidos.

System calls (“syscalls”)

- Carregar registradores ou variáveis com parâmetros necessários
  - Instrução INT (“trap”)
    - Gera uma interrupção (de software)
      - Mudança automática para modo “kernel”
- Parâmetros dizem ao S.O. o que fazer
- No seu término
  - “Retorna” como se fosse uma chamada à função
  - Força o retorno ao modo “usuário”

System calls: linux

Linux: *getpid*() in the standard C library looks like this:

```
0x400b65e0 <getpid+0>:  mov    $0x14,%eax
0x400b65e5 <getpid+5>:  int    $0x80
0x400b65e7 <getpid+7>:  cmp    $0xfffff001,%eax
0x400b65ec <getpid+12>: jae    0x400b65ef <getpid+15>
0x400b65ee <getpid+14>: ret
0x400b65ef <getpid+15>: ...  ; Error handling
0x400b660c <getpid+44>: jmp    0x400b65ee <getpid+14>
```

0x80 (128) is the system call interrupt number, 0x14 (20) means getpid.

Return value of the system call is stored in the `%eax` register.

In Linux: an unsigned number larger than 0xfffff001 means error.

In 2's complement, it means a negative number in the range -4095 – -1.

The code above is called the C-library glue to system call.

So that the users don't need to know how to make INTs themselves.

Resumo

- A program calls the C-library function like *getpid*().
- *getpid*() puts the service number (20) into the register `%eax`, and generates a software interrupt using `int $0x80`.
- The interrupt vector specifies that the kernel function `system_call` should be executed in kernel mode.
- `system_call` checks that the service number is valid, and looks up a table to find and execute the implementation function *sys_getpid*().
- *sys_getpid*() finds the PID and puts the result to `%eax` and returns to `system_call`.
- `system_call` uses `IRET` to get back to the user program, and the CPU falls back to user mode.
- *getpid*() checks for error and returns to the user program.

Cadeia de chamadas para fork() em Linux

- libc fork()
- system_call (arch/i386/kernel/entry.S)
- sys_clone() (arch/i386/kernel/process.c)
- do_fork() (kernel/fork.c)
- copy_process() (kernel/fork.c)
- p = dup_task_struct(current) // shallow copy
- copy_* // copy point-to structures
- copy_thread () // copy stack, regs, and eip
- wake_up_new_task() // set child runnable

Cadeia de chamadas para exit() em Linux

- libc exit(code)
- system_call (arch/i386/kernel/entry.S)
- sys_exit() (kernel/exit.c)
- do_exit() (kernel/exit.c)
- exit_*() // free data structures
- exit_notify() // tell other processes we exit
  // reparent children to init
  // EXIT_ZOMBIE
  // EXIT_DEAD

Exercício

- Em Linux, o número da interrupção que dá acesso à syscalls é a 0x80 ($128).
  - E em xv6 ?
  - No Windows ?
- Usando o código-fonte do xv6 ou o debugger (gdb), liste todas as chamadas em cascata decorrente das syscalls fork() e exit().