CS353 Linux Kernel

Chentao Wu吴晨涛
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

上海交通大學

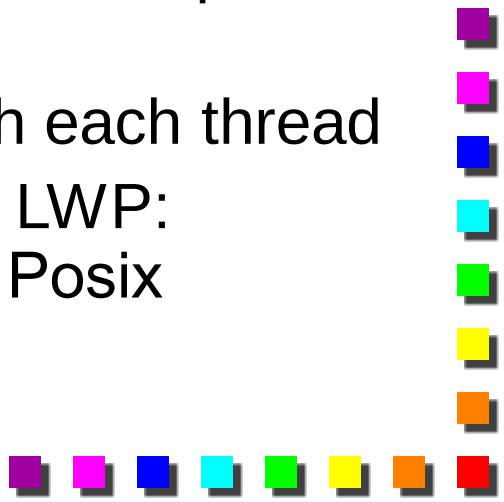
3A. Process Management -- Introduction

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

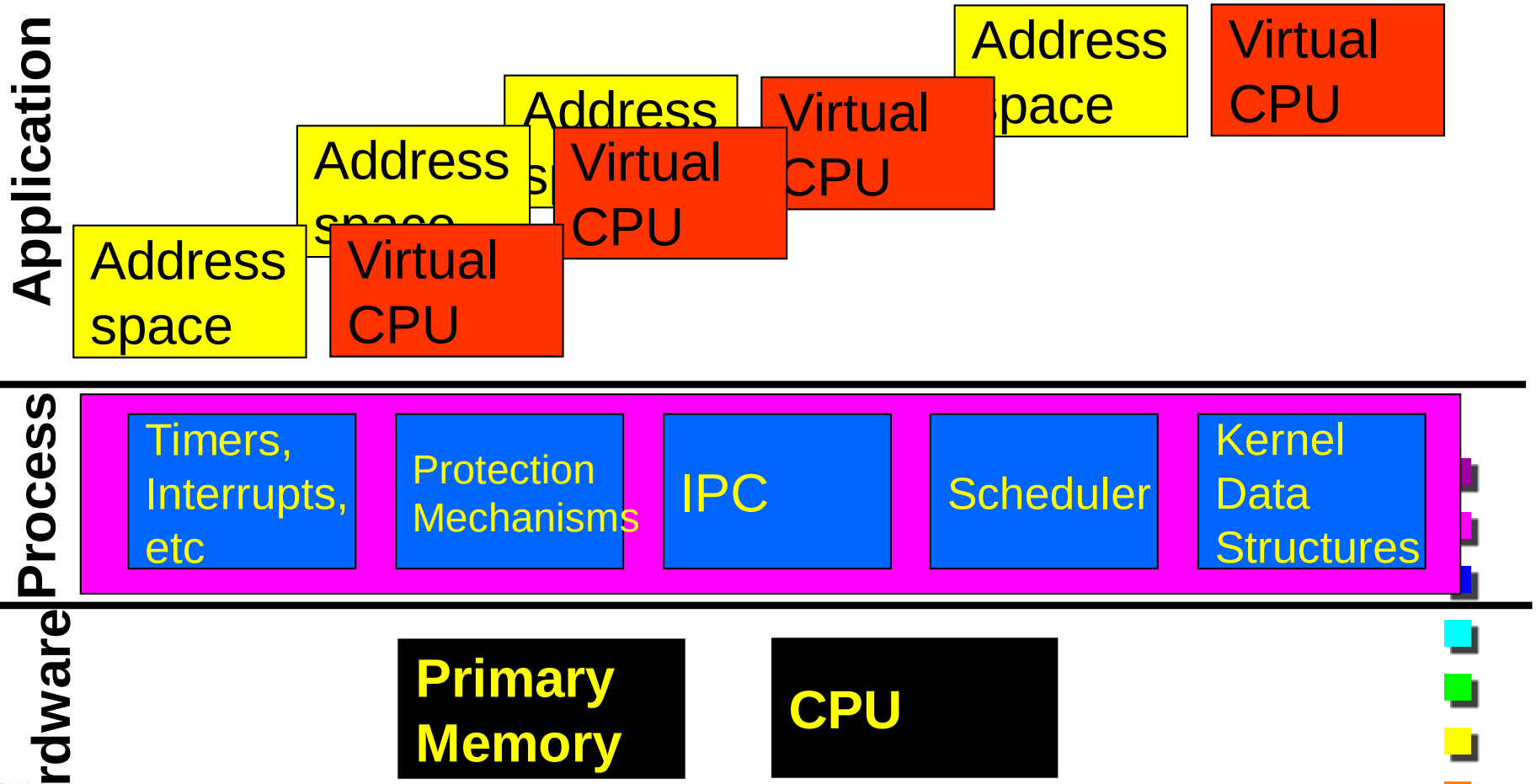
上海交通大學

Processes, Lightweight Processes, and Threads

- *Process*: an instance of a program in execution
- *(User) Thread*: an execution flow of the process
  - Pthread (POSIX thread) library
- *Lightweight process (LWP)*: used to offer better support for multithreaded applications
  - LWP may share resources: address space, open files, …
  - To associate a lightweight process with each thread
  - Examples of pthread libraries that use LWP: LinuxThreads, IBM's Next Generation Posix Threading Package (NGPT)

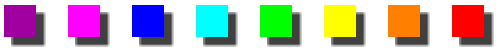


Process Management

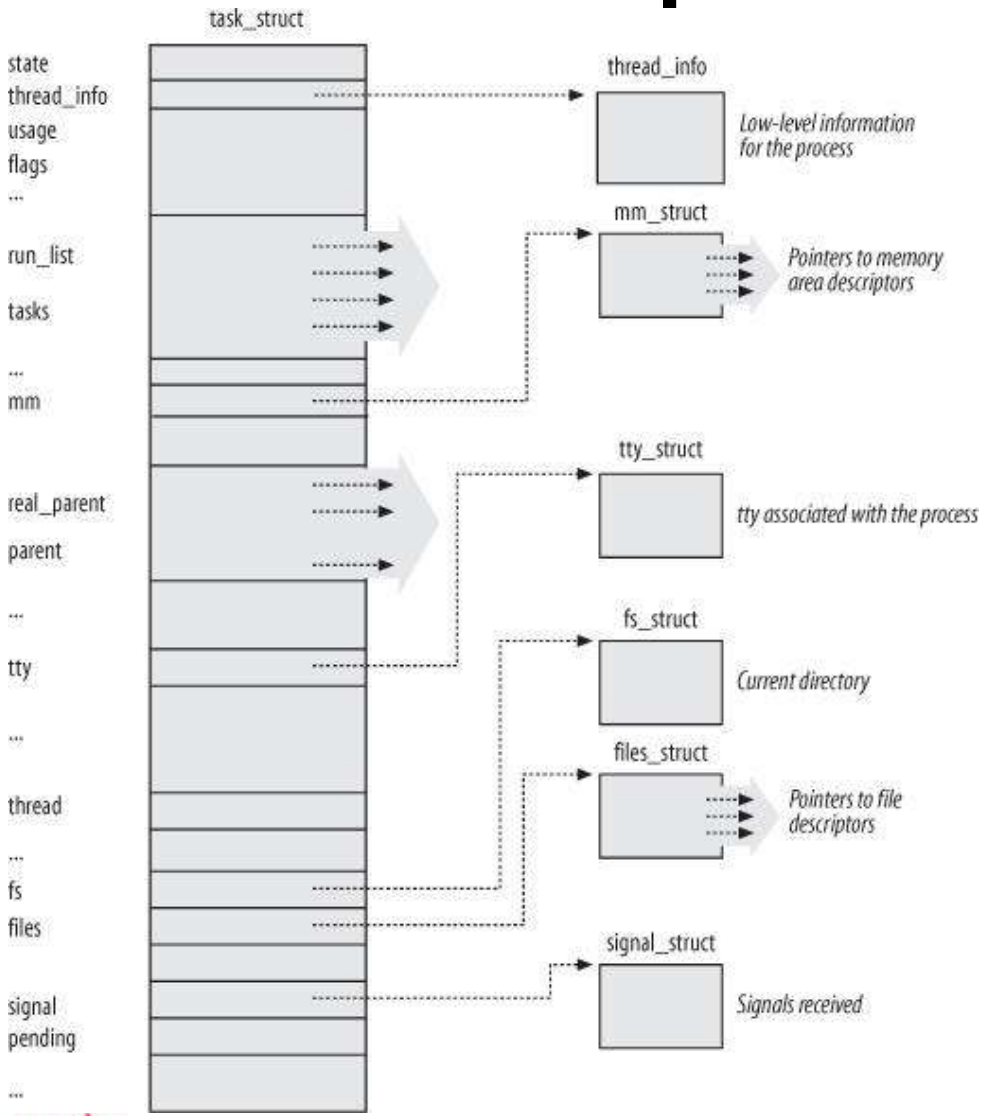


Process Descriptor

- *task_struct* data structure
  - state: process state
  - thread_info: low-level information for the process
  - mm: pointers to memory area descriptors
  - tty: tty associated with the process
  - fs: current directory
  - files: pointers to file descriptors
  - signal: signals received
  - …

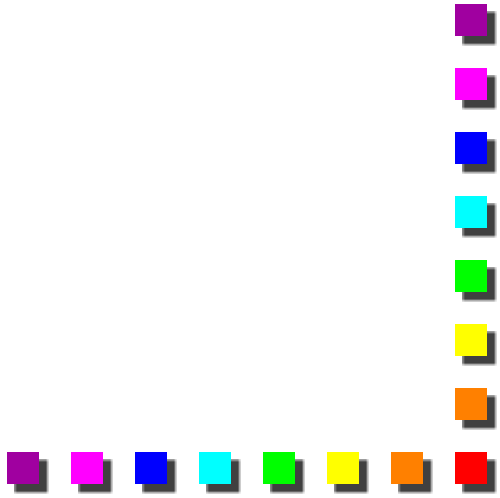


Linux Process Descriptor



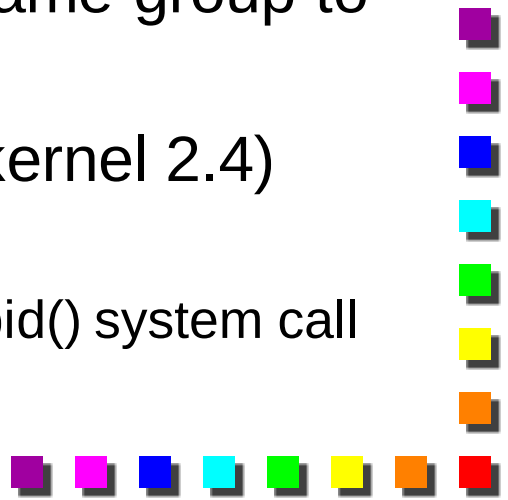
Process State

- TASK_RUNNING: executing
- TASK_INTERRUPTABLE: suspended (sleeping)
- TASK_UNINTERRUPTABLE: (seldom used)
- TASK_STOPPED
- TASK_TRACED
- EXIT_ZOMBIE
- EXIT_DEAD



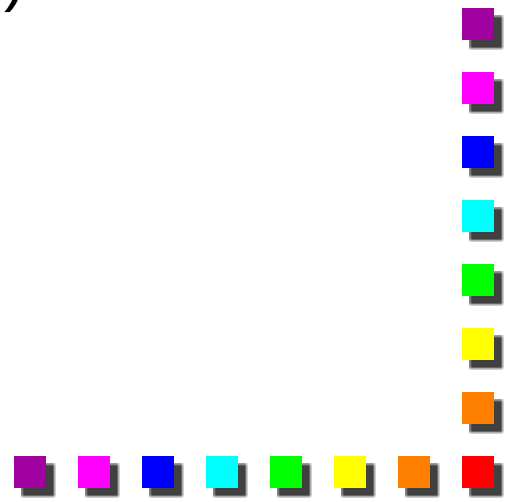
Identifying a Process

- Process descriptor pointers: 32-bit
- Process ID (PID): 16-bit (~32767 for compatibility)
  - Linux associates different PID with each process or LWP
  - Programmers expect threads in the same group to have a common PID
  - *Thread group*: a collection of LWPs (kernel 2.4)
    - The PID of the first LWP in the group
    - *tgid* field in process descriptor: using getpid() system call

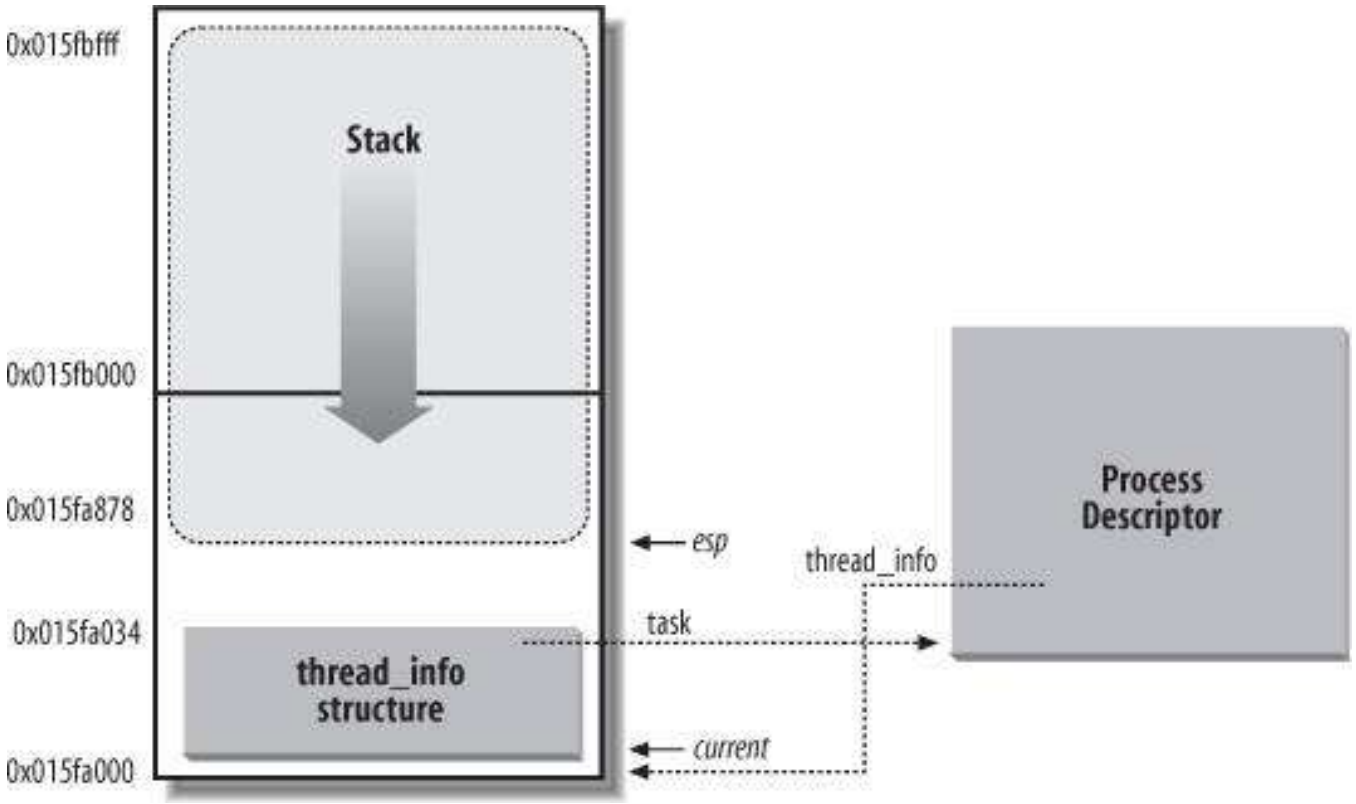


Process Descriptor Handling

- union thread_union {
  struct thread_info thread_info;
  unsigned long stack[2048];
  };
- Two data structures in 8KB (2 pages)
  - *thread_info* structure (new to 3rd ed.)
  - Kernel mode process stack



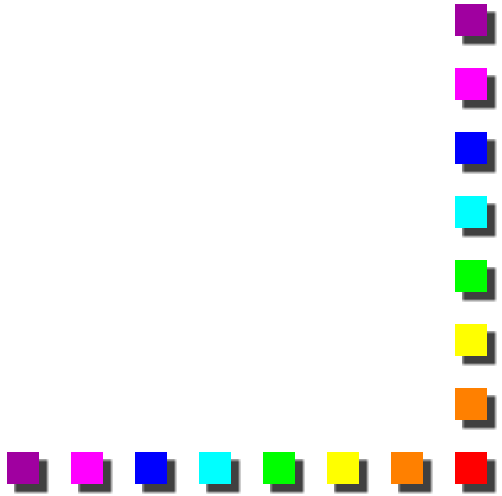
Storing the *thread_info* Structure and the Process Kernel Stack



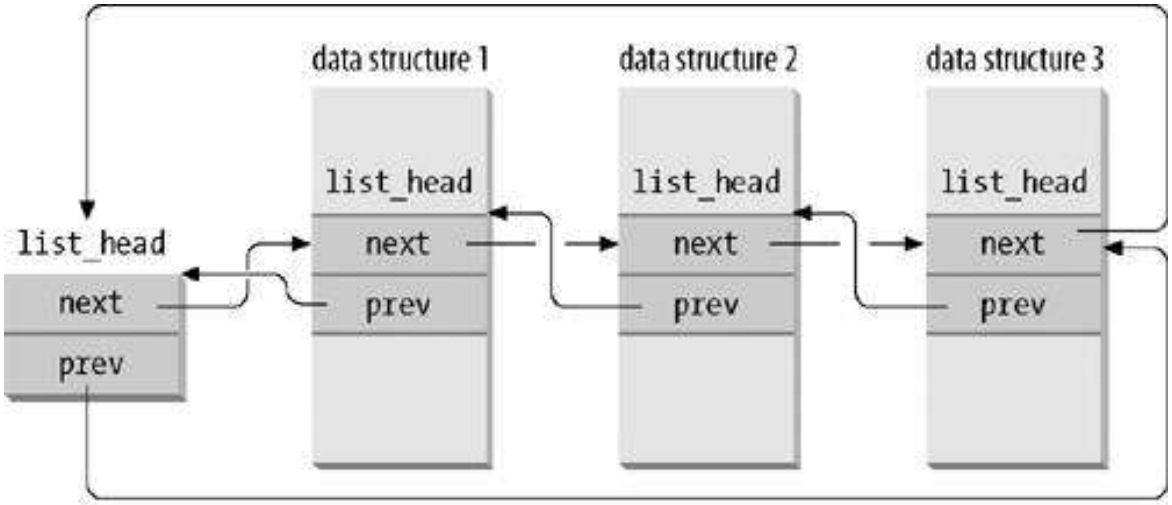
上海交通大學

Identifying the Current Process

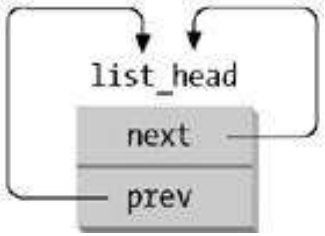
- Obtain the address of *thread_info* structure from the *esp* register
  - current_thread_info()
  - movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
- The process descriptor pointer of the process currently running on a CPU
  - The *current* macro: equivalent to current_thread_info()->task
  - movl $0xffffe000, %ecx
andl %esp, %ecx
movl (%ecx), p



Doubly Linked Lists Built with the *list_head* Data Structure



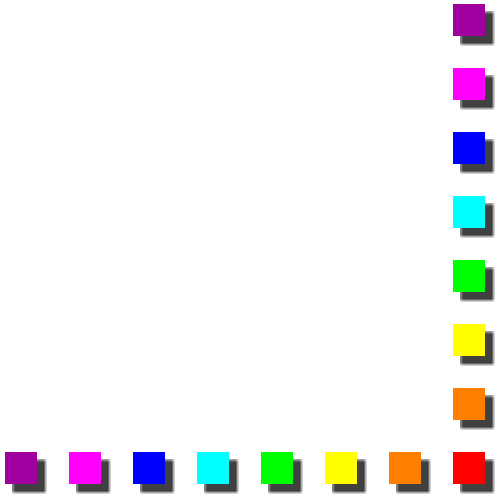
(a) a doubly linked listed with three elements



(b) an empty doubly linked list

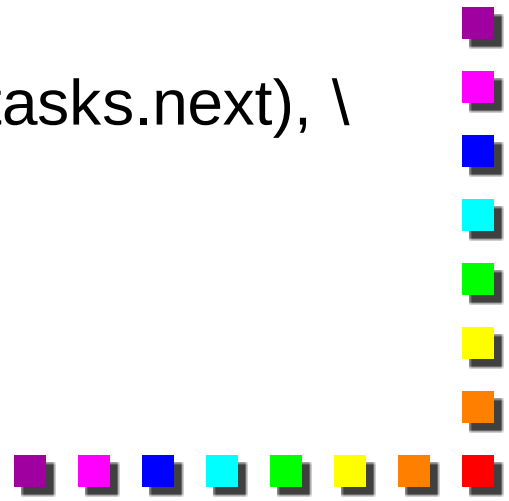
List Handling Functions and Macros

- LIST_HEAD(list_name)
- list_add(n,p)
- list_add_tail(n,p)
- list_del(p)
- list_empty(p)
- list_entry(p,t,m)
- list_for_each(p,h)
- list_for_each_entry(p,h,m)



The Process List

- *tasks* field in *task_struct* structure
  - type *list_head*
  - *prev*, *next* fields point to the previous and the next *task_struct*
- Process 0 (swapper): *init_task*
- Useful macros:
  - SET_LINKS, REMOVE_LINKS: insert and remove a process descriptor
  - #define for_each_process(p) \
    for (p=&init_task; (p=list_entry((p)->tasks.next), \
    struct task_struct, tasks) \
) != &init_task;)

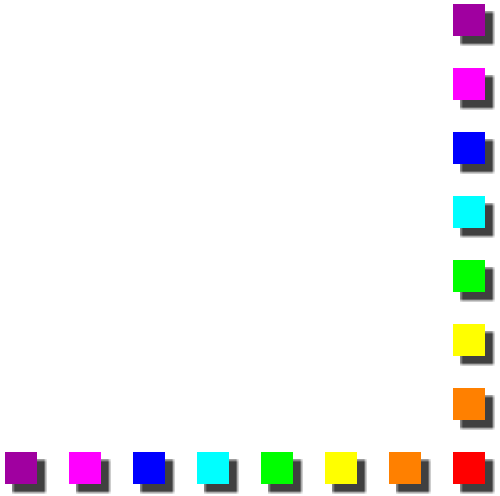


List of TASK_RUNNING processes

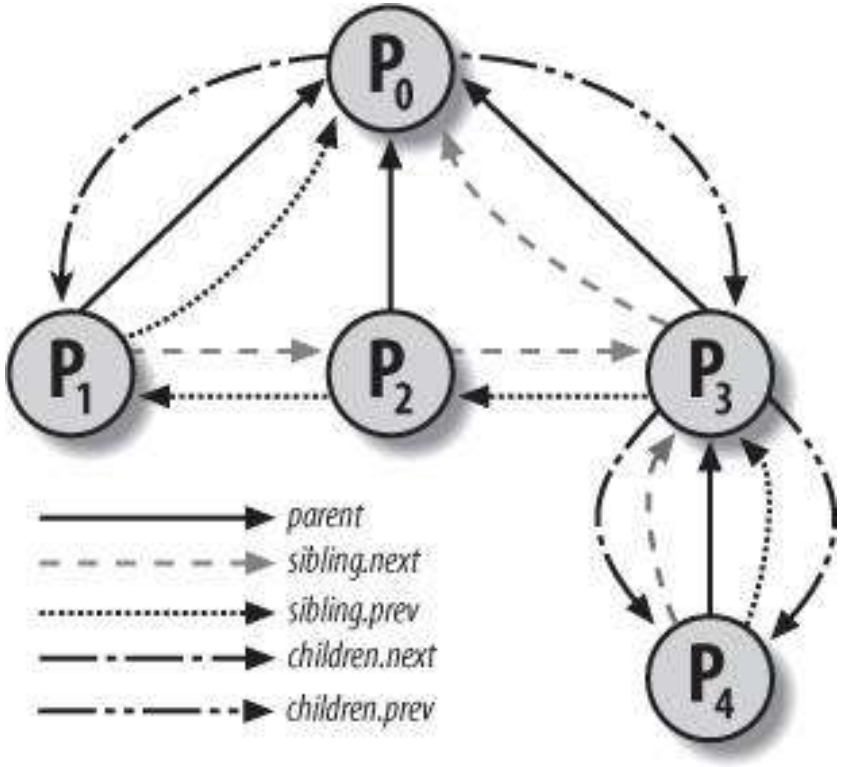
- *runqueue*
  - *run_list* field in task_struct structure: type list_head
- Linux 2.6 implements the runqueue differently
  - To achieve scheduler speedup, Linux 2.6 splits the runqueue into 140 lists of each priority!
  - *array* filed of process descriptor: pointer to the *prio_array_t* data structure
    - nr_active: # of process descriptors in the list
    - bitmap: priority bitmap
    - queue: the 140 list_heads
  - enqueue_task(p, array), dequeue_task(p, array)

Parenthood Relationships among Processes

- Process 0 and 1: created by the kernel
  - Process 1 (*init*): the ancestor of all processes
- Fields in process descriptor for parenthood relationships
  - real_parent
  - parent
  - children
  - sibling



Parenthood Relationships among Five Processes



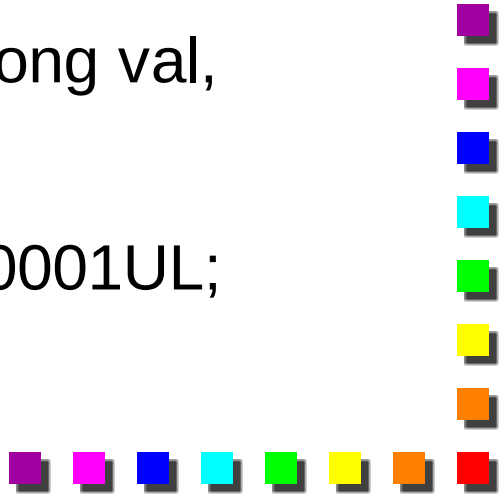
Pidhash Table and Chained Lists

- To search up the search for the process descriptor of a PID
  - Sequential search in the process list is inefficient
- The *pid_hash* array contains four hash tables and corresponding filed in the process descriptor
  - pid: PIDTYPE_PID
  - tgid: PIDTYPE_TGID (thread group leader)
  - pgrp: PIDTYPE_PGID (group leader)
  - session: PIDTYPE_SID (session leader)
- *Chaining* is used to handle PID *collisions*

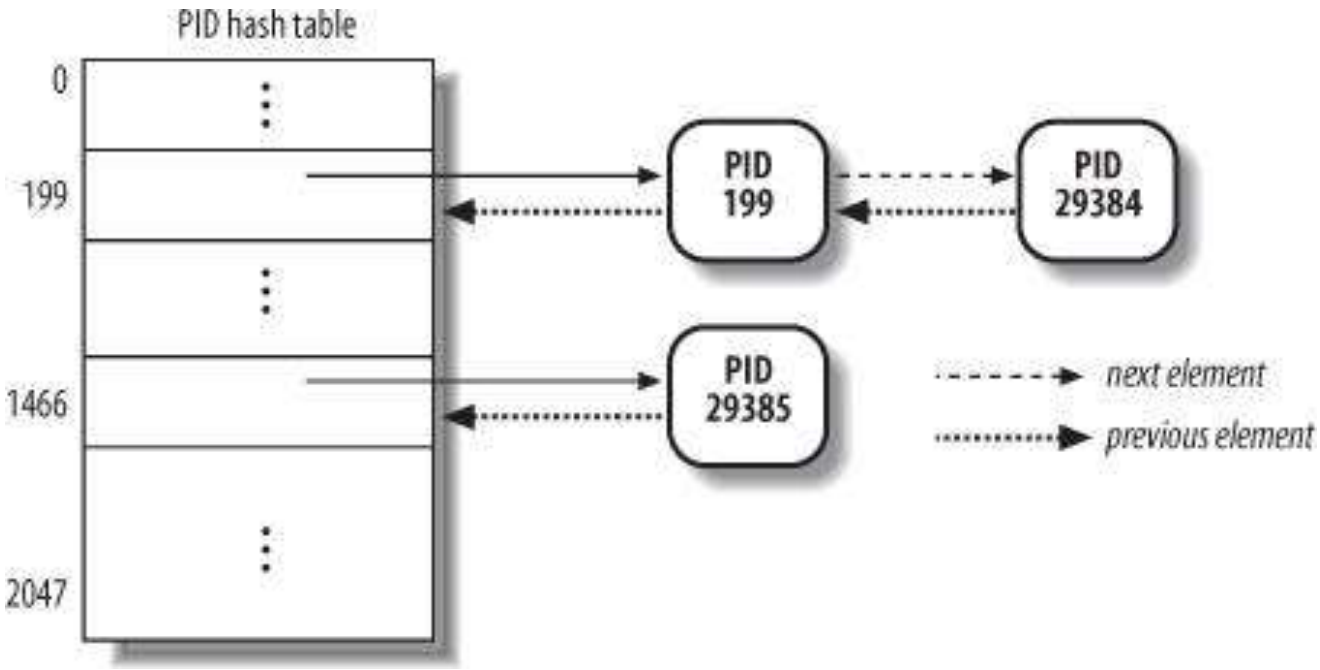
The pidhash Table

- Size of each pidhash table: dependent on the available memory
- PID is transformed into table index using pid_hashfn macro
  - #define pid_hashfn(x) hash_long((unsigned long)x, pidhash_shift)
  - unsigned long hash_long(unsigned long val, unsigned int bits)

```
{
  unsigned long hash = val * 0x9e370001UL;
  return hash >> (32-bits);
}
```



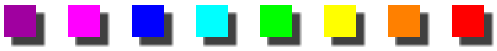
A Simple Example PID Hash Table and Chained Lists



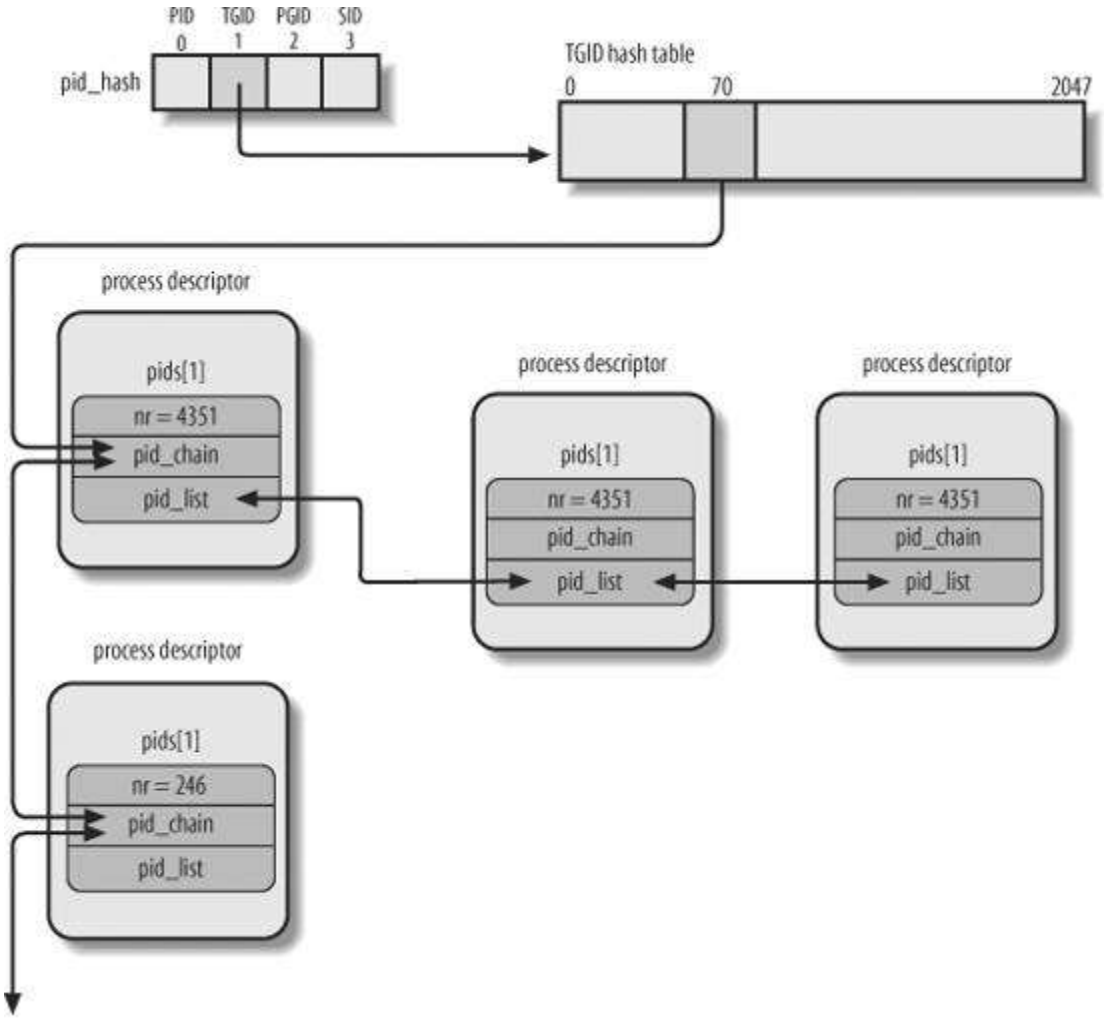
上海交通大學

PID

- *pids* field of the process descriptor: the pid data structures
  - nr: PID number
  - pid_chain: links to the previous and the next elements in the hash chain list
  - pid_list: head of the per-PID list (in thread group)

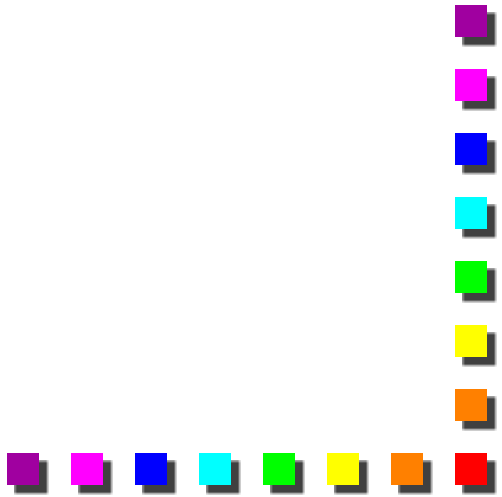


The PID Hash Tables



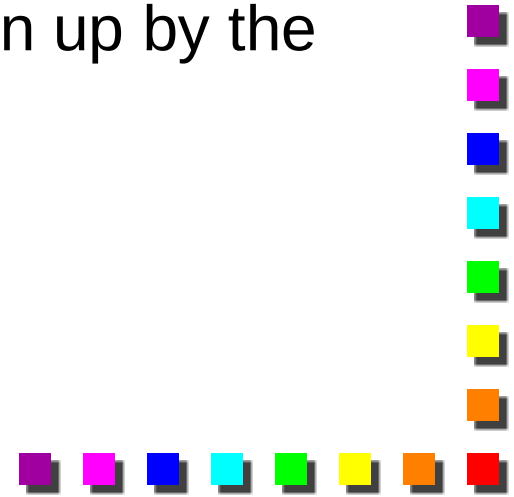
PID Hash Table Handling Functions and Macros

- do_each_trask_pid(nr, type, task)
- while_each_trask_pid(nr, type, task)
- find_trask_by_pid_type(type, nr)
- find_trask_by_pid(nr)
- attach_pid(task, type, nr)
- detach_pid(task, type)
- next_thread(task)



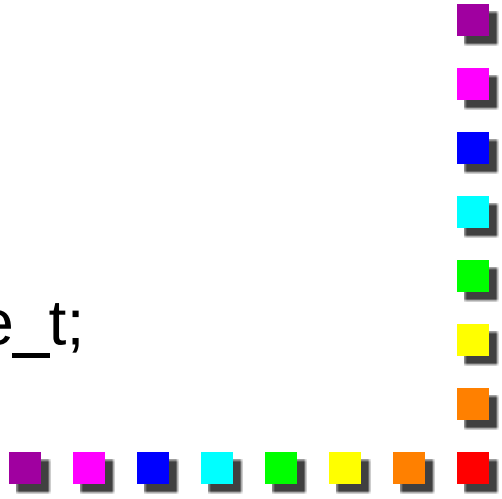
How Processes are Organized

- Processes in TASK_STOPPED, EXIT_ZOMBIE, EXIT_DEAD: not linked in lists
- Processes in TASK_INTERRUPTABLE, TASK_UNINTERRUPTABLE: *wait queues*
- Two kinds of sleeping processes
  - *Exclusive process*
  - *Nonexclusive process*: always woken up by the kernel when the event occurs



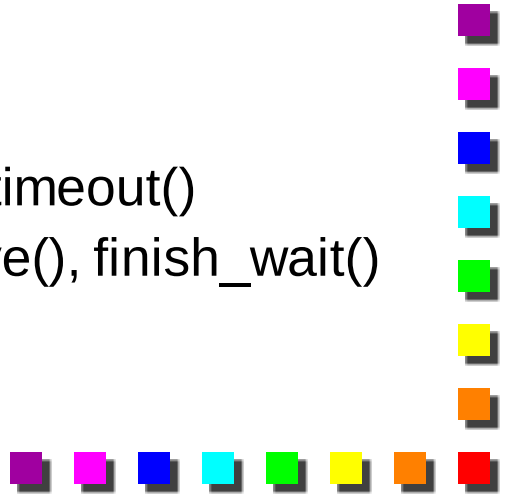
Wait Queues

- struct _ _wait_queue_head {
  spinlock_t lock;
  struct list_head task_list;
  };
  typedef struct _ _wait_queue_head
  wait_queue_head_t;
- struct _ _wait_queue {
  unsigned int flags;
  struct task_struct * task;
  wait_queue_func_t func;
  struct list_head task_list;
  };
  typedef struct _ _wait_queue wait_queue_t;



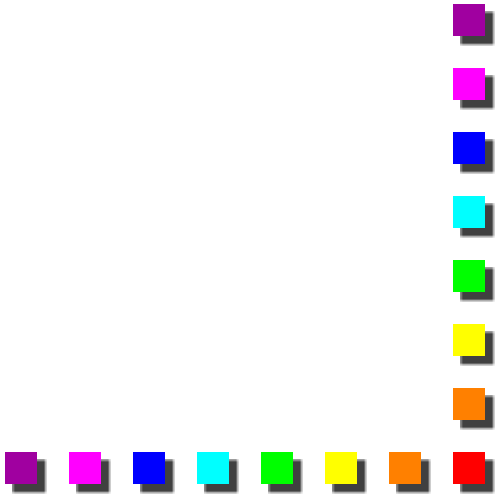
Handling Wait Queues (1)

- Wait queue handling functions:
  - add_wait_queue()
  - add_wait_queue_exclusive()
  - remove_wait_queue()
  - wait_queue_active()
  - DECLARE_WAIT_QUEUE_HEAD(name)
  - init_waitqueue_head()
- To wait:
  - sleep_on()
  - interruptible_sleep_on()
  - sleep_on_timeout(), interruptible_sleep_on_timeout()
  - Prepare_to_wait(), prepare_to_wait_exclusive(), finish_wait()
  - Macros: wait_event, wait_event_interruptible



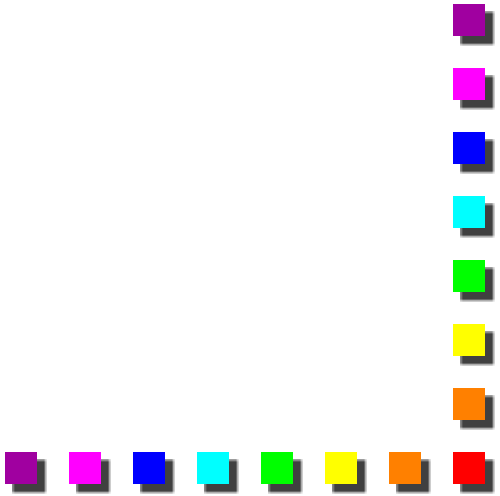
Handling Wait Queues (2)

- To be woken up:
  - Wake_up, wake_up_nr, wake_up_all, wake_up_sync, wake_up_sync_nr, wake_up_interruptible, wake_up_interruptible_nr, wake_up_interruptible_all, wake_up_interruptible_sync, wake_up_interruptible_sync_nr



Process Resource Limits

- RLIMIT_AS
- RLIMIT_CORE
- RLIMIT_CPU
- RLIMIT_DATA
- RLIMIT_FSIZE
- RLIMIT_LOCKS
- RLIMIT_MEMLOCK
- RLIMIT_MSGQUEUE
- RLIMIT_NOFILE
- RLIMIT_NPROC
- RLIMIT_RSS
- RLIMIT_SIGPENDING
- RLIMIT_STACK

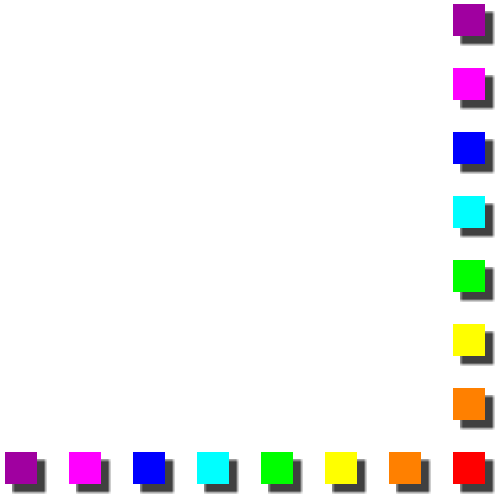


Process Switch

- Process switch, task switch, *context switch*
  - *Hardware* context switch: a far jmp (in older Linux)
  - *Software* context switch: a sequence of mov instructions
    - It allows better control over the validity of data being loaded
    - The amount of time required is about the same
- Performing the Process Switch
  - Switching the Page Global Directory
  - Switching the Kernel Mode stack and the hardware context

Task State Segment

- TSS: a specific segment type in x86 architecture to store hardware contexts

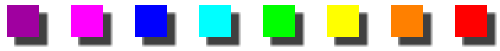


Creating Processes

- In traditional UNIX, resources owned by parent process are duplicated
  - Very slow and inefficient
- Mechanisms to solve this problem
  - Copy on Write: parent and child read the same physical pages
  - Lightweight process: parent and child share per-process kernel data structures
  - vfork() system call: parent and child share the memory address space

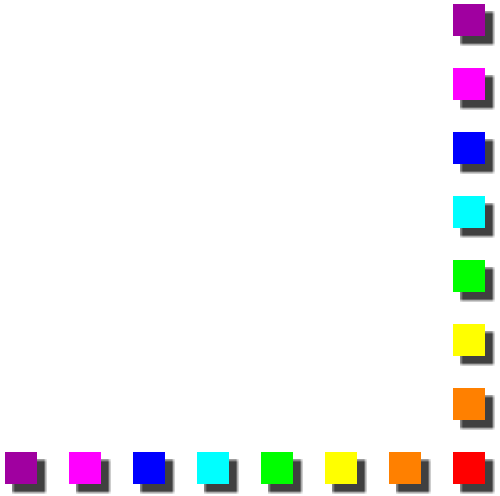
clone(), fork(), and vfork() System Calls

- clone(fn, arg, flags, child_stack, tls, ptid, ctid): creating lightweight process
  - A wrapper function in C library
  - Uses clone() system call
- *fork()* and *vfork()* system calls: implemented by clone() with different parameters
- Each invokes *do_fork()* function



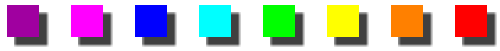
Kernel Threads

- Kernel threads run only in kernel mode
- They use only linear addresses greater than PAGE_OFFSET



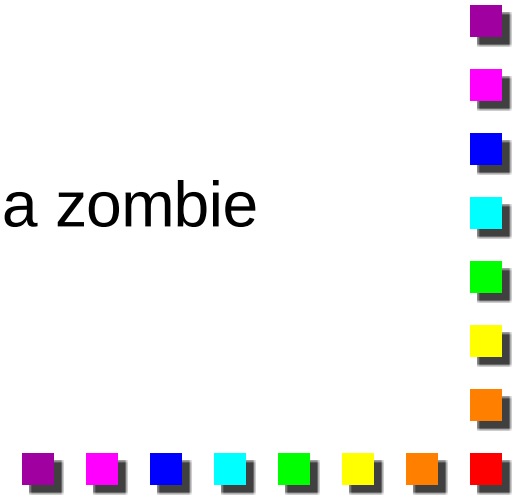
Kernel Threads

- kernel_thread(): to create a kernel thread
- Example kernel threads
  - Process 0 (swapper process), the ancestor of all processes
  - Process 1 (init process)
  - Others: keventd, kapm, kswapd, kflushd (also bdflush), kupdated, ksoftirqd, …
  -



Destroying Processes

- exit() library function
  - Two system calls in Linux 2.6
    - _exit() system call
      - Handled by *do_exit()* function
    - exit_group() system call
      - By *do_group_exit()* function
- Process removal
  - Releasing the process descriptor of a zombie process by release_task()



Project 2B:
Process Management

上海交通大學

Process: schedule in times

- **Add ctx, a new member to task_struct to record the schedule in times of the process;**
  - **When a task is scheduled in to run on a CPU, increase ctx of the process;**
  - **Export ctx under /proc/XXX/ctx;**

- More detailed information is shown in Student Experimental Handbook.