

Process Creation, Termination, and Management

James Tsai, 25 Mar 2002

NCTU CIS Operating System lab.

2002 Linux kernel trace seminar

Outline

- I Introduction
 - Reference Counting
 - Process Structure
 - Process Descriptor
 - Relations between Processes
 - Pid Hash table
- I Process Creation
 - __clon(), fork(), vfork()
 - do_fork()
- I Process Termination
 - exit(), sys_exit()
 - do_exit()

Processes and Threads

- | A process is usually defined as an instance of a program in execution.
- | Threads are separate contexts of execution within the same process.
- | Threads are processes that happen to share the same global memory space, but they have different stacks.
- | In Linux process and thread have a more generic name – task
- | Each process has its own **process descriptor**
- | Linux is a multi-task OS

Reference Counting

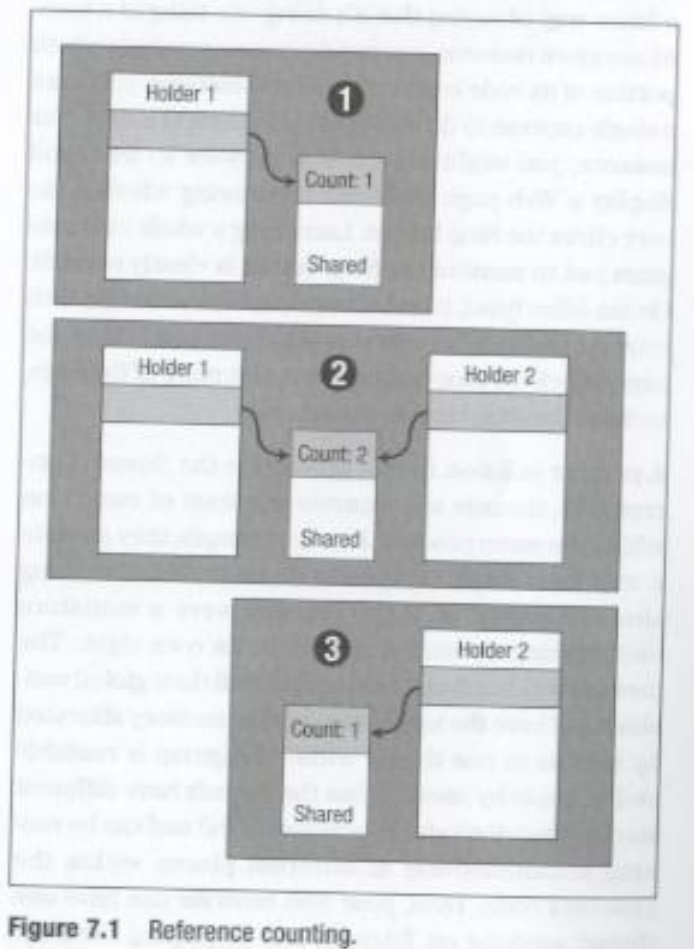


Figure 7.1 Reference counting.

- I In general terms, one or more “holder” object carry a pointer to a shared data object that includes an integer called it reference count
- I The value of this reference count equals the number of holder objects share the data

Process ID (PID)

- | Any Unix-like operating system allows users to identify process by means of a number called the Process ID (PID)
- | The PID is a 32-bit unsigned integer stored in the `pid` field of process descriptor
- | However, for compatibility with traditional Unix systems developed for 16-bit hardware platforms, the maximum PID number allowed on Linux is 32767
- | When the kernel creates the 32768th process in the system, it must start recycling the lower unused PIDs.

Process store in memory

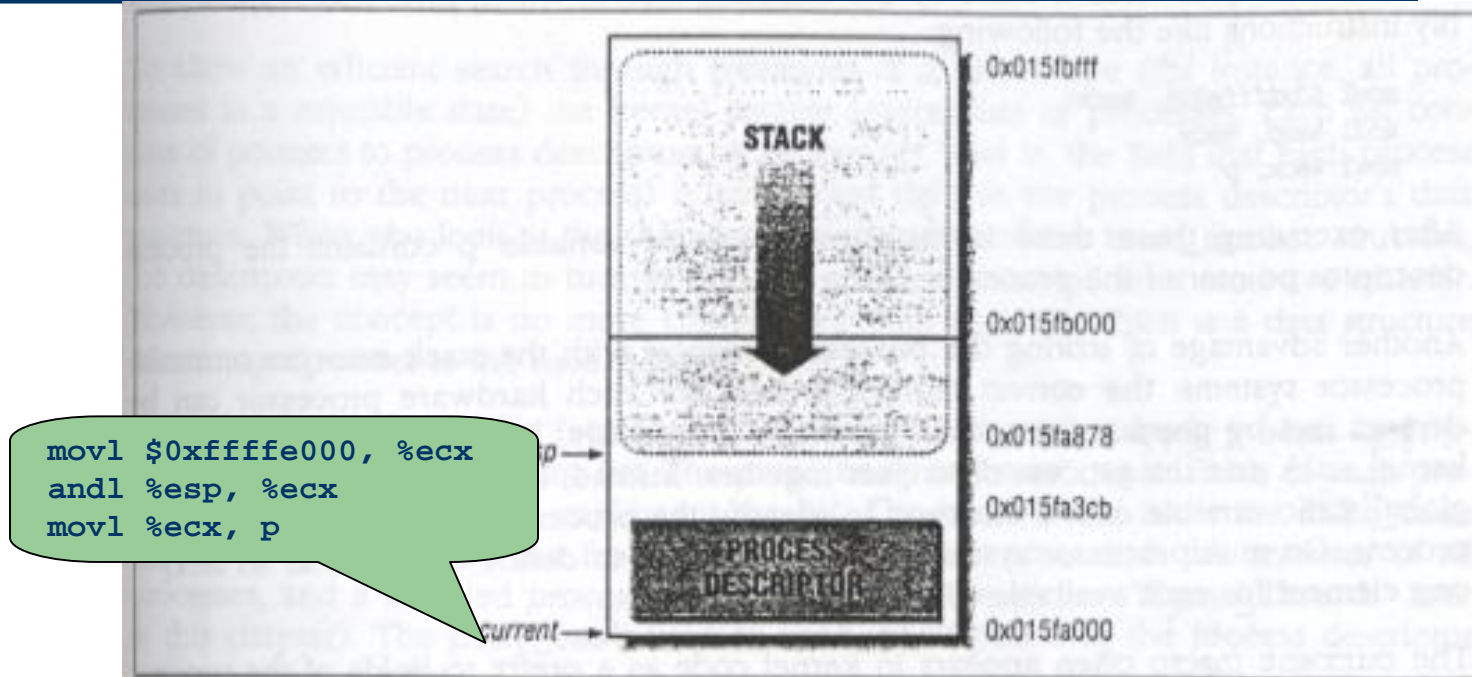


Figure 3-2. Storing the process descriptor and the process kernel stack

- I Linux stores two different data structures for each process in a single **8KB** memory area: the **process descriptor** and **Kernel Mode process stack**
- I **esp** register is the CPU stack pointer, which is used to address the stack's top location

Process Descriptor

struct task_struct() (1)

/include/linux/sched.h

```
struct task_struct {
    ....
    volatile long state;
    unsigned long flags;
    ....
    struct list_head run_list;
    ....
    struct task_struct *next;
    ....
    pid_t pid;
    ....
    struct task_struct *p_opptr, *p_pptr, *p_opptr, *p_yptr, *p_osptr,
    ....
    struct task_struct *pidhash_next;
    struct task_struct **pidhash_pprev;
    ....
}
```

Process State

#define TASK_RUNNING 0
The process is waiting to be executed

#define TASK_INTERRUPTIBLE 1
The process is suspended until some condition becomes true, okay to be interrupted by a signal

#define TASK_UNINTERRUPTIBLE 2
The process is suspended until some hardware condition becomes true, not to be interrupted by a signal

#define TASK_ZOMBIE 4
The process execution is terminated, but the parent process has not yet issued a wait()-like system call to return information about the dead process

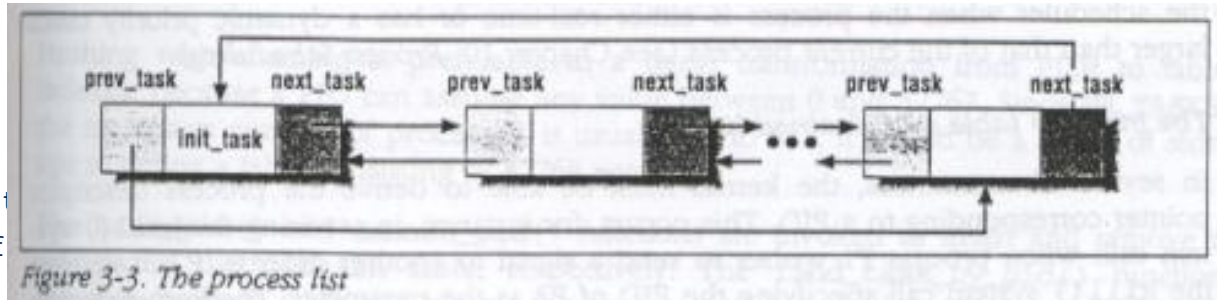
#define TASK_STOPPED 8
The process is being stopped by signal or tracing

Process Descriptor

struct task_struct() (2)

[/include/linux/sched.h](#)

```
struct task_struct {  
    ....  
    volatile long s  
    unsigned long f  
    ....  
    struct list_head run_list;  
    ....  
    struct task_struct *next_task, *prev_task;  
    ....  
    pid_t pid;  
    ....  
    struct task_struct *p_opptr, *p_pptr, *p_cptra, *p_ysptr, *p_osptr;  
    ....  
    struct task_struct *pidhash_next;  
    struct task_struct **pidhash_pprev;  
    ....  
}
```



Process Descriptor

struct task_struct() (?)

/include/linux/sche

```

struct task_str
....
volatile
unsigned
....
struct
....
struct
....
pid_t p
....
....
struct task_struct *p_opptr, *p_pptr, *p_cptra, *p_ysptr, *p_osptr;
....
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;
....
}
    
```

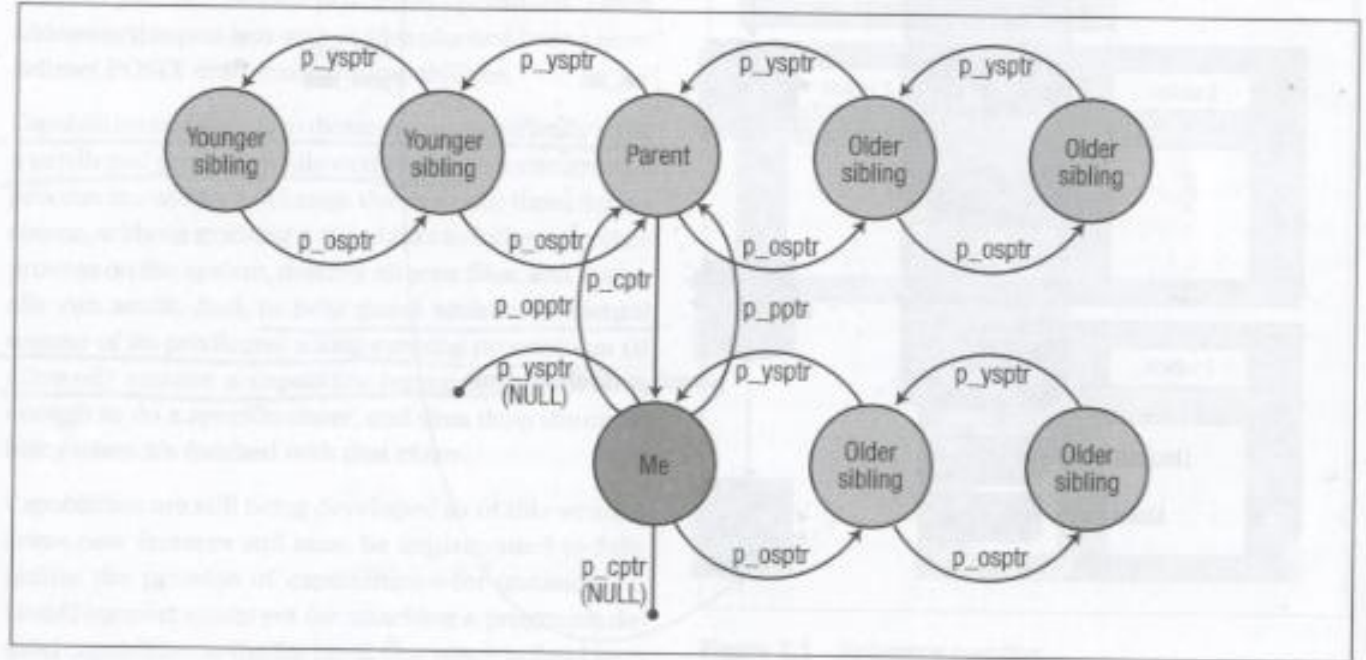


Figure 7.3 The process graph.

Process Descriptor

struct task_struct() (4)

/include/linux/sched.h

```
struct task_struct {
    ....
    volatile long state;
    unsigned long flags;
    ....
    struct list_head run_list;
    ....
    struct task_struct *next;
    ....
    pid_t pid;
    ....
    struct task_struct *p_opport;
    ....
    struct task_struct *pidhash[16];
    struct task_struct **pidhash_ptr[16];
    ....
}
```

/include/linux/list.h

```
struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(amp;name),
    &name }

-----
-----
```

/include/linux/sched.h

```
#define INIT_TASK(tsk) \
    ....
    run_list:
    LIST_HEAD_INIT(tsk.run_list),
    \
}
```

Process De *struct ta*

/include/linux/sched.h

```
struct task_struct {
    ....
    volatile long state
    unsigned long flags
    ....
    struct list_head run
    ....
    struct task_struct
    ....
    pid_t pid;
    ....
    struct task_struct
    ....
    struct task_struct
    struct task_struct
    ....
}
```

/kernel/sched.c

```
static inline void add_to_runqueue(struct
task_struct * p)
{
    list_add(&p->run_list, &runqueue_head);
    nr_running++;
}
static inline void move_last_runqueue(struct
task_struct * p)
{
    list_del(&p->run_list);
    list_add_tail(&p->run_list, &runqueue_head);
}
static inline void move_first_runqueue(struct
task_struct * p)
{
    list_del(&p->run_list);
    list_add(&p->run_list, &runqueue_head);
}
```

/include/linux/sched.h

```
static inline void del_from_runqueue(struct
task_struct * p)
{
    nr_running--;
    p->sleep_time = jiffies;
    list_del(&p->run_list);
    p->run_list.next = NULL;
}
static inline int task_on_runqueue(struct
task_struct *p)
{
    return (p->run_list.next != NULL);
}
```

Process Descriptor

struct task_struct() (5)

/include/linux/sched.h

```
struct task_struct {
    ....
    volatile long state;
    unsigned long flags;
    ....
    struct list_head run_list;
    ....
    struct task_struct *next_task, *prev_task;
    ....
    pid_t pid;
    ....
    struct task_struct *p_opptr, *p_pptr, *p_cptra, *p_ysptr, *p_osptr;
    ....
    struct task_struct *pidhash_next;
    struct task_struct **pidhash_pprev;
    ....
}
```

pidhash table with chained list

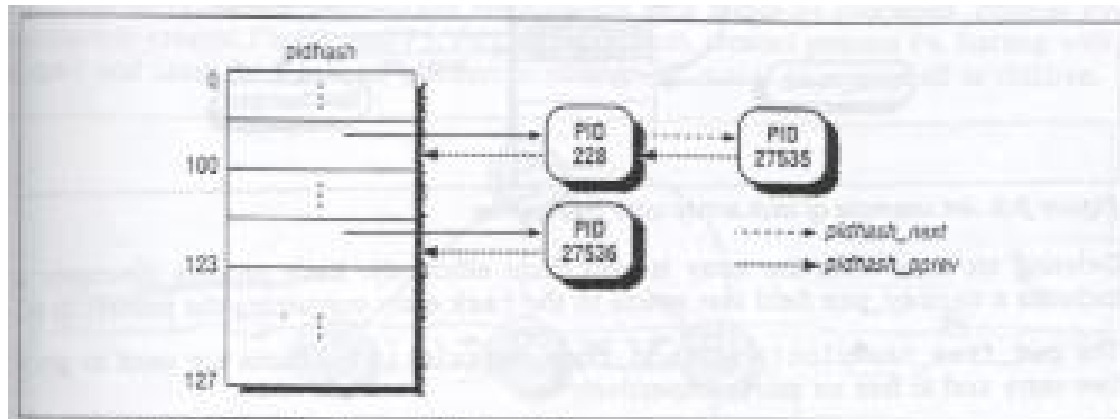


Figure 3-4. The pidhash table and chained lists.

- ```
#define PIDHASH_SZ (4096 >> 2)
#define pid_hashfn(x) (((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1)
```
- **pidhash\_next**
  - **pidhash\_pprev**
  - **hash\_pid(), unhash\_pid()**
  - **Find\_task\_by\_pid()**

What

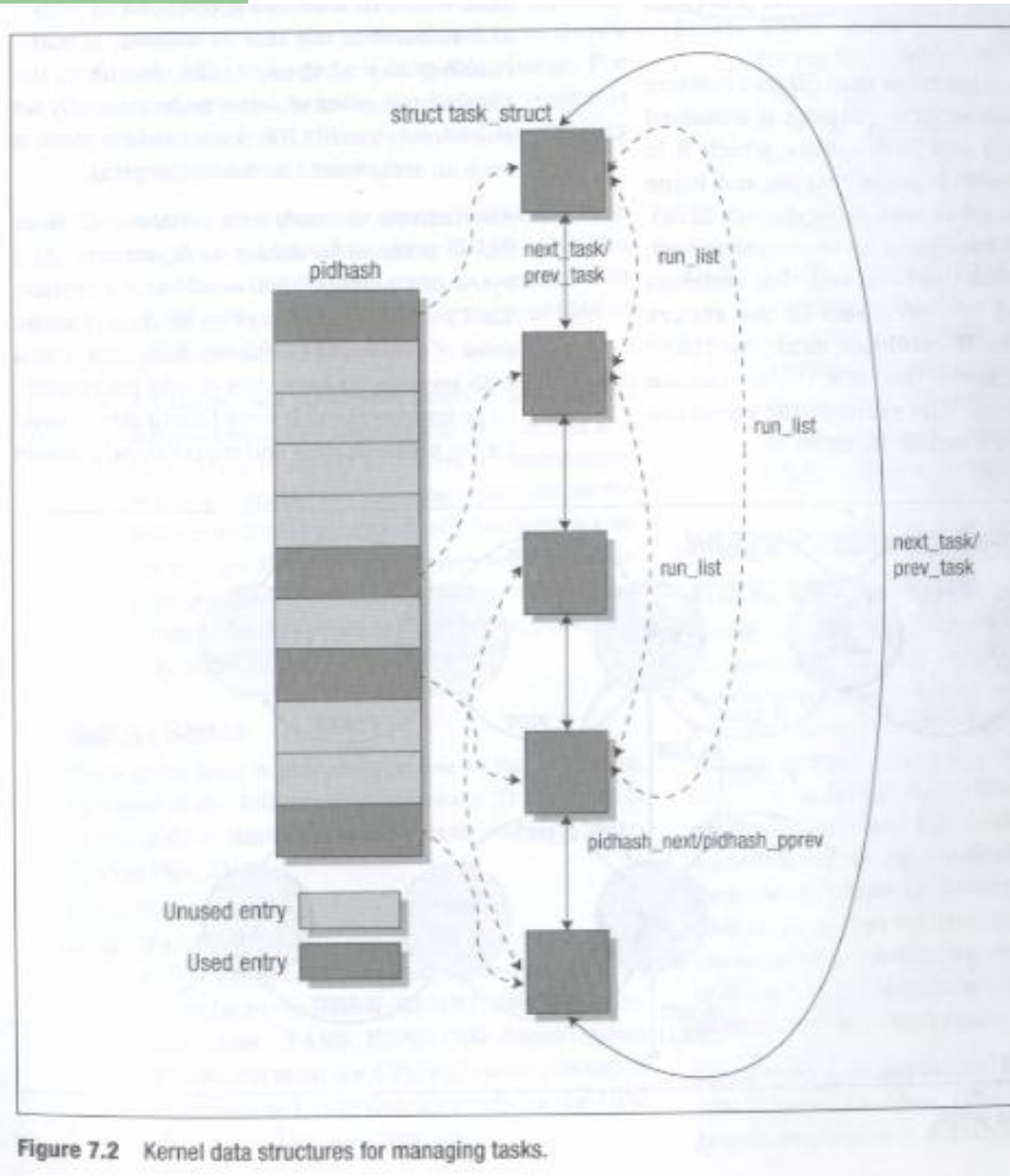


Figure 7.2 Kernel data structures for managing tasks.

A large green shape on the left side of the page, featuring a white semi-circular cutout on its right edge. A dark blue horizontal bar with rounded ends extends from the bottom right of the green shape towards the right edge of the page.

# Process Creation

# Where Processes Come From: `__clone()`

- I Use to create lightweight processes
- I A wrapper function defined in `unistd.h`
- I Four parameters:
  - **fn**: Specify a function to be executed in the child process.
  - **arg**: Pointer to data passed to the child process.
  - **flags**: the low byte of the flags specifies the signal that terminates; The remaining bits are flags which specify the resources shared between the parent and the child.
  - **child\_stack**: Specifies the User Mode stack pointer to be assigned to the esp register of the child process. If = 0, assign the current parent stack pointer.

Generally in the low byte is `SIGCHLD` signal

Clone flags (in `/sched.h`):

```
#define CLONE_VM 0x00000100
#define CLONE_FS 0x00000200
#define CLONE_FILES 0x00000400
#define CLONE_SIGHAND 0x00000800
#define CLONE_PID 0x00001000
#define CLONE_PTRACE 0x00002000
#define CLONE_VFORK 0x00004000
#define CLONE_PARENT 0x00008000
#define CLONE_THREAD 0x00010000
```

# Where Processes Come From: *clone(), fork(), vfork()*

- I `__clone()` makes use of a Linux system call hidden to the programmer – `clone()`
- I `clone()` only receives the `flags` and `child_stack` parameters
- I `fork()` is implemented by Linux as a `clone()`
  - (`SIGCHLD` signal and all clone flags cleared, 0)
- I `vfork()` is implemented by Linux as a `clone()`
  - (`SIGCHLD` signal and `CLONE_VM` and `CLONE_VFORK`, 0)
- I When these system calls are issued, the kernel invokes the `do_fork()` function

# Where Processes Come From: *do\_fork( ) (1)*

/kernel/fork.c

```
int do_fork(unsigned long clone_flags, unsigned long stack_start,
 struct pt_regs *regs, unsigned long stack_size)
{
 int retval;
 struct task_struct *p;
 struct completion vfork;

 retval = -EPERM;

 if (clone_flags & CLONE_PID) {
 if (current->pid)
 goto fork_out;
 }

 retval = -ENOMEM;
 p = alloc_task_struct();
 if (!p)
 goto fork_out;

 *p = *current;

}
```

/\*  
\* CLONE\_PID is only allowed for the initial SMP swapper  
\* calls  
\*/

# Where Processes Come From: *do\_fork*

## /kernel/fork.c

```
int do_fork(unsigned long clone_flags,
 struct pt_regs *regs, unsigned long stack_size)
{
 int retval;
 struct task_struct *p;
 struct completion vfork;

 retval = -EPERM;

 if (clone_flags & CLONE_F
 if (current->pid
 goto fork_out;
 }

 retval = -ENOMEM;
 p = alloc_task_struct();
 the new process */
 if (!p)
 goto fork_out;

 *p = *current;

}
```

## /include/asm-i386/processor.h

```
#define alloc_task_struct()
 ((struct task_struct *)
 __get_free_pages(GFP_KERNEL,1))


```

## /mm/page\_alloc.c

```
unsigned long __get_free_pages(unsigned
int gfp_mask, unsigned int order)
{
 struct page * page;

 page = alloc_pages(gfp_mask,
order);
 if (!page)
 return 0;
 return (unsigned long)
page_address(page);
}
```

resent

# Where Processes Come From: *do\_fork( ) (3)*

/kernel/fork.c

```
int do_fork(unsigned long clone_flags, unsigned long stack_start,
 struct pt_regs *regs, unsigned long stack_size)
{
 int retval;
 struct task_struct *p;
 struct completion vfork;

 retval = -EPERM;

 if (clone_flags & CLONE_PARENT)
 if (current->pid)
 goto fork_out;
}

retval = -ENOMEM;
p = alloc_task_struct();
if (!p)
 goto fork_out;

*p = *current;

.....
```

/include/asm-i386/current.h

```
#define current get_current()

get_current(void)
{
 struct task_struct *current;
 __asm__("andl %%esp,%0; : "=r"
(current) : "" (~8191UL));
 return current;
}
```

# Where Processes Come From: *do\_fork()* (4)

/kernel/fork.c

```
.....
p->pid = get_pid(clone_flags);
p->run_list.next = NULL;
p->run_list.prev = NULL;
.....
/* CLONE_PARENT and CLONE_THREAD re-use the old parent
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & (CLONE_PARENT | CLONE_THREAD))) {
 p->p_opptr = current;
 if (!(p->ptrace & PT_PTRACED))
 p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
 p->tgid = current->tgid;
 list_add(&p->thread_group, ¤t->thread_group);
}

SET_LINKS(p);
hash_pid(p);
nr_threads++;
.....
wake_up_process(p); /* do this last */
++total_forks;
if (clone_flags & CLONE_VFORK)
 wait_for_completion(&vfork);
```

Initially, the new process is not placed in the **run queue**. It's still possible that `do_fork` will fail, and it would be wasteful to put the new process in the run queue.

More seriously, the new process is not completely initialized yet, and we wouldn't want another CPU to hand control this process prematurely.

Normally, `do_fork`'s caller should be registered as the parent of the new process.

When clone flags == `CLONE_PARENT` or `CLONE_THREAD`, this means that the new process should have the same parent as `do_fork`'s caller.

If the calling process is not being traced, the caller is also made the new process's logical parent – the one to which signal notifications are sent.

However, if the caller is being traced, the child's logical parent will remain the same as its parent's logical parent, the debug process.

# Where Processes Come From: *do\_fork( ) (5)*

/kernel/fork.c

```
....
p->pid = get_pid(clone_flags);
p->run_list.next = NULL;
p->run_list.prev = NULL;
....
/* CLONE_PARENT and CLONE_THREAD re-use the old element */
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & (CLONE_PA
 p->p_opptr = current;
 if (!(p->ptrace & PT
 p->p_pptr = c
})
if (clone_flags & CLONE_THREA
 p->tgid = current->
 list_add(&p->thr
)

SET_LINKS(p);
hash_pid(p);
nr_threads++;
....
wake_up_process(p); /* do this last */
++total_forks;
if (clone_flags & CLONE_VFORK)
 wait_for_completion(&vfork);
```

/include/linux/sched.h

```
#define SET_LINKS(p) do { \
 (p)->next_task = &init_task; \
 (p)->prev_task = init_task.prev_task; \
 init_task.prev_task->next_task = (p); \
 init_task.prev_task = (p); \
 (p)->p_ysptr = NULL; \
 if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL) \
 (p)->p_osptr->p_ysptr = p; \
 (p)->p_pptr->p_cptr = p; \
} while (0)
```

# Where Processes Come From: *do\_fork( ) (6)*

/kernel/fork.c

```
.....
p->pid = get_pid(clone_flags);
p->run_list.next = NULL;
p->run_list.prev = NULL;
.....
/* CLONE_PARENT and CLONE_CHILD_CLEAREV */
p->p_opptr = current->opptr;
p->p_pptr = current->pptr;
if (!(clone_flags & CLONE_PARENT))
 p->p_opptr = current->opptr;
if (!(p->ptraced))
 p->p_pptr = current->pptr;
}
if (clone_flags & CLONE_PARENT)
 p->tgid = current->tgid;
list_add(&p->thread_list, ¤t->thread_list);
}
```

include/linux/sched.h

```
#define pid_hashfn(x) (((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1)

static inline void hash_pid(struct task_struct *p)
{
 struct task_struct **htable = &pidhash[pid_hashfn(p->pid)];

 if((p->pidhash_next = *htable) != NULL)
 (*htable)->pidhash_pprev = &p->pidhash_next;
 *htable = p;
 p->pidhash_pprev = htable;
}
```

```
SET_LINKS(p);
hash_pid(p); /* enters the new process into the pidhash table by calling hash_pid */
nr_threads++;
.....
wake_up_process(p);
++total_forks;
if (clone_flags & CLONE_VFORK)
 wait_for_completion(&vfork);
```

Put the new process in the **TASK\_RUNNING** state and enters it into the run queue via call `wake_up_process`.

# Where Processes Come From: *do\_fork( ) (7)*

/kernel/fork.c

```
....
p->pid = get_pid(clone_flags);
p->run_list.next = NULL;
p->run_list.prev = NULL;
....
/* CLONE_PARENT and CLONE_THREAD re-use the old parent */
p->p_opptr = current->p_opptr;
p->p_pptr = current->p_pptr;
if (!(clone_flags & (CLONE_PARENT | CLONE_THREAD))) {
 p->p_opptr = current;
 if (!(p->ptrace & PT_PTRACED))
 p->p_pptr = current;
}
if (clone_flags & CLONE_THREAD) {
 p->tgid = current->tgid;
 list_add(&p->thread_group, ¤t->thread_group);
}

SET_LINKS(p);
hash_pid(p); /* enters the new process into the pidhash table by calling hash_pid */
nr_threads++;
....
wake_up_process(p); /* do this last */
++total_forks;
if (clone_flags & CLONE_VFORK)
 wait_for_completion(&vfork);
```

# Where Processes Come From

- | Not only is the `struct task_struct` now filled out, but all the relevant kernel data structures – task-list, run-queue, parent-child-relationship, and PID hash table – have all been correctly modified to account for new arrival.
- | But now you only make copies of same process over and over. So after this, you must call `exec` family.
- | In Linux kernel, it implements all functions in the `exec` family is `do_execve( )`.

# PID Allocation

## *get\_pid()* (1)

/kernel/fork.c

```
static int get_pid(unsigned long flags)
{
 static int next_safe = PID_MAX;
 struct task_struct *p;

 if (flags & CLONE_PID)
 return current->pid;

 if(++last_pid & 0xffff8000) {
 last_pid = 300;
 goto inside;
 }

}
```

/include/linux/threads.h

```
#define PID_MAX 0x8000
```

Low numbered PIDs tend to belong to long-running daemons that are created at startup.

# PID Allocation

## *get\_pid()* (2)

/kernel/fork.c

```
if(last_pid >= next_safe) {
inside:
 next_safe =
 read_lock(&tasklist_lock);
repeat:
 for_each_task(p) {
 if(p->pid == last_pid ||
 p->pgrp == last_pid ||
 p->tgid == last_pid ||
 p->session == last_pid) {
 if(++last_pid >= next_safe) {
 if(last_pid & 0xffff8000)
 last_pid = 300;
 next_safe = PID_MAX;
 }
 goto repeat;
 }
 if(p->pid > last_pid && next_safe > p->pid)
 next_safe = p->pid;

 }
 read_unlock(&tasklist_lock);
}
spin_unlock(&lastpid_lock);

return last_pid;
}
```

/include/linux/sched.h

```
#define for_each_task(p) \
 for (p = &init_task ; (p = p->next_task) != \
 &init_task ;)
```

If a new task is being created and the only available PIDs are below 300, this loop will simply continue until some process with a higher-numbered PID exits. On a UP system, the kernel will never exit this loop, so it will never hand the CPU to any process.

# Process Termination

# Where processes come to an end *exit()*, *sys\_exit()*, *do\_exit()*

- | Process could terminate voluntarily by invoke the system call **exit()**
- | Or, when a C program returns from **main**, it implicitly calls **exit()**
- | **exit()** is implemented in the kernel with **sys\_exit()** function
- | **sys\_exit()**'s job is to turn a living process into a zombie
- | You can also kill a process by sending it signal SIGKILL
- | **do\_exit()** doesn't immediately deallocate the **struct task\_struct**

/kernel/exit.c

```
asmlinkage long sys_exit(int error_code)
{
 do_exit((error_code&0xff)<<8);
}
```

**sys\_exit()** simply converts the exit code into the format that **do\_exit()** expects, and then call **do\_exit()**

# Where processes come to an end

## *do\_exit( ) (1)*

### /kernel/exit.c

```
NORET_TYPE void do_exit(long code)
{
 struct task_struct *tsk = current;

 if (in_interrupt())
 panic("Aiee, killing interrupt handler!");
 if (!tsk->pid)
 panic("Attempted to kill the idle task!");
 if (tsk->pid == 1)
 panic("Attempted to kill init!");
 tsk->flags |= PF_EXITING;
 del_timer_sync(&tsk->real_timer);

fake_volatile:
#ifdef CONFIG_BSD_PROCESS_ACCT
 acct_process(code);
#endif

}
```

# Where pro d

/kernel/exit.c

```
.....
 __exit_mm(tsk);

 lock_kernel();
 sem_exit();
 __exit_files(tsk);
 __exit_fs(tsk);
 exit_sighand(tsk);
 exit_thread();

 if (current->leader)
 disassociate_ctty(1);

 put_exec_domain(tsk->exec_domain);
 if (tsk->binfmt && tsk->binfmt->module)
 __MOD_DEC_USE_COUNT(tsk->binfmt->module);

 tsk->exit_code = code;
 exit_notify();
 schedule();
 BUG();

 goto fake_volatile;
}
```

Free its allocated memory

```
static inline void __exit_mm(struct task_struct * tsk)
{
 struct mm_struct * mm = tsk->mm;

 mm_release();
 if (mm) {
 atomic_inc(&mm->mm_count);
 if (mm != tsk->active_mm) BUG();
 /* more a memory barrier than a real lock */
 task_lock(tsk);
 tsk->mm = NULL;
 task_unlock(tsk);
 enter_lazy_tlb(mm, current, smp_processor_id());
 mmput(mm);
 }
}
```

| Free its semaphores and other V IPC structure.

| Releases its **allocated files**

| Release its **file system data**

| Release **signal handler table**

The exiting task's exit code is remembered for future use by its parent

# Where processes come to an end

## *do\_exit()* (3)

/kernel/exit.c

```
.....
__exit_mm(tsk);

lock_kernel();
sem_exit();
__exit_files(tsk);
__exit_fs(tsk);
exit_sighand(tsk);
exit_thread();

if (current->leader)
 disassociate_ctty(1);

put_exec_domain(tsk->exec_domain);
if (tsk->binfmt && tsk->binfmt->module)
 __MOD_DEC_USE_COUNT++;
 __MOD_DEC_USE_COUNT--;
 tsk->binfmt->module);

tsk->exit_code = code;
exit_notify();
schedule();
BUG();

goto fake_volatile;
}
```

Puts the process in the **TASK\_ZOMBIE** state and then alerts the exiting task's parent and the members of its process group to their compatriot's demise.

kernel/exit.c

```
.....
current->exit_signal = SIGCHLD;
.....
write_lock_irq(&tasklist_lock);
current->state = TASK_ZOMBIE;
do_notify_parent(current, current->exit_signal);
.....
```

The call to **schedule()** never returns, because it switches context to another process and never switches back.