

# Ch 3. Process: The Principal Model of Execution

Oct. 31, 2006

Hyun-koo Jee

Senior researcher, DWE

# Contents

3.0. Intro.

3.1. Introducing Our Program

3.2. Process Descriptor

3.3. Process Creation: `fork()`, `vfork()`, & `clone()`

3.4. Process Lifespan

3.5. Process Termination

3.6. Keeping Track of Process: Basic Scheduler  
Construction

3.7. Wait Queues

3.8. Asynchronous Execution Flow

# Contents

## **3.0. Intro.**

3.1. Introducing Our Program

3.2. Process Descriptor

3.3. Process Creation: `fork()`, `vfork()`, & `clone()`

3.4. Process Lifespan

3.5. Process Termination

3.6. Keeping Track of Process: Basic Scheduler  
Construction

3.7. Wait Queues

3.8. Asynchronous Execution Flow

# 3.0. Intro.

- Process
  - Definition: an instance of a program execution
  - $\text{process} = \text{running program (core image)} + \text{PCB (process control block)}$
  - $\text{process} = \text{thread} + \text{address space}$   
(Each process has an independent address space.)
  - , “ load ”
    - <Ctrl-alt-del> + <alt-t> in Windows
    - ‘ps’ or ‘top’ in UNIX/LINUX
- Thread
  - Definition: serial execution stream
  - LWP (LightWeight Process)
  - 1 thread uses 1 virtual CPU.

# 3.0. Intro.

- Each process has its own...
    - text (code)
    - data (global vars, static vars)
    - stack
    - heap
- ... in memory.

## 3.0. Intro.

- Each process has its own...
  - Address Space
  - File Descriptor Table
  - Signal Handler Table
  - ...

# 3.0. Intro.

- User Mode & Kernel Mode

\* OS :

<i>Application</i>	<i>H.W. resource</i>
_____	_____

# 3.0. Intro.

- User Mode & Kernel Mode

\* , CPU ,  
User Mode Kernel Mode .

- User Mode

- Application 가
- H.W. resource \_\_\_\_\_ ( )

- Kernel Mode

- Kernel  
( , interrupt, trap, system call )
- H.W. resource \_\_\_\_\_

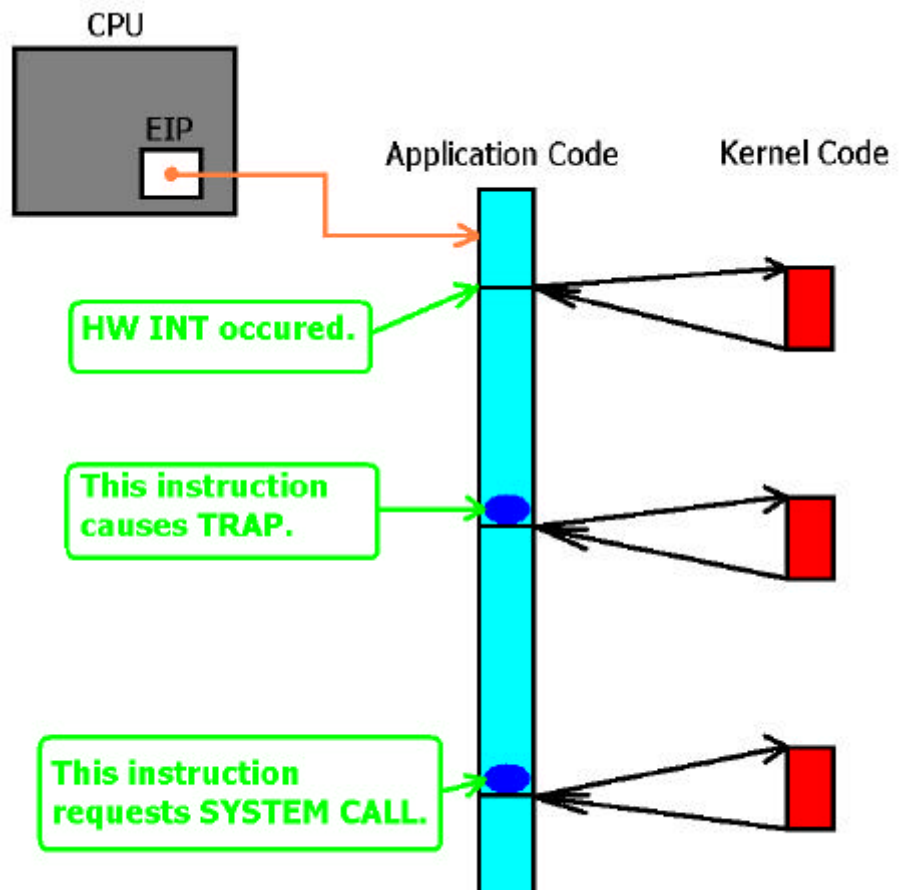
\* “privilege levels” or “rings” in x86



# 3.0. Intro.

- User Mode & Kernel Mode

*If there is  
only 1 process...*



# 3.0. Intro.

- User Mode & Kernel Mode

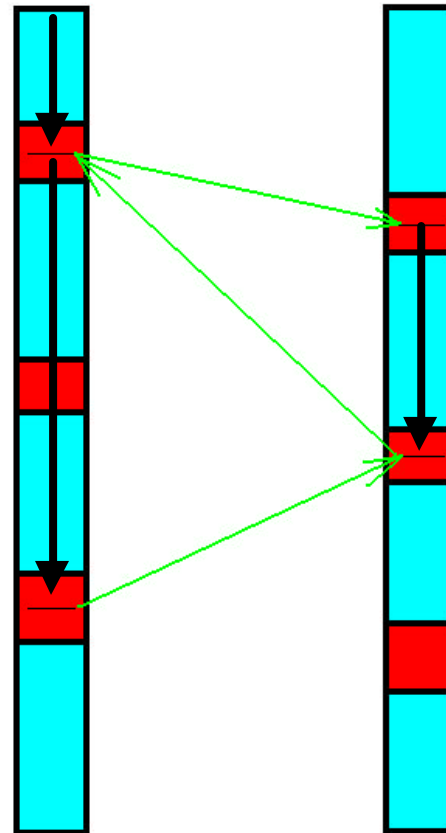
*If there is  
only 1 process...*



# 3.0. Intro.

- Context Switching

*If there are  
2 processes...*



# 3.0. Intro.

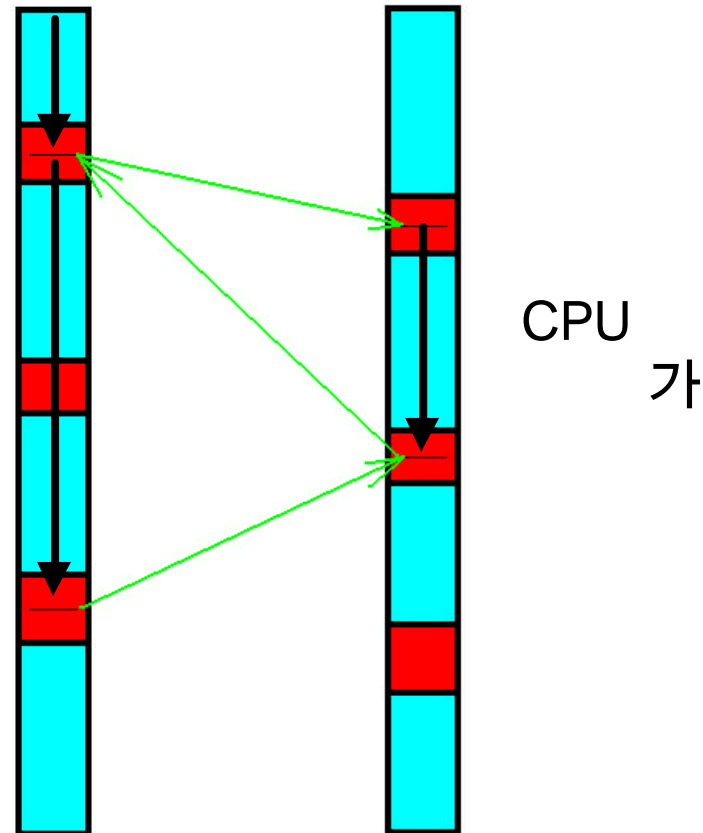
- Context Switching

*If there are  
2 processes...*

**“CONTEXT  
SWITCHING”**

**“must save the context”**

**“1 ProcessControlBlock  
per process”**



# 3.0. Intro.

- Note:

Shell is not a part of kernel.

It's just an user-mode application.

ex) bash, tcsh, ... in *UNIX/LINUX*  
explorer.exe in *Windows*  
COMMAND.COM in *MS-DOS*

# Contents

3.0. Intro.

## **3.1. Introducing Out Program**

3.2. Process Descriptor

3.3. Process Creation: fork(), vfork(), & clone()

3.4. Process Lifespan

3.5. Process Termination

3.6. Keeping Track of Process: Basic Scheduler  
Construction

3.7. Wait Queues

3.8. Asynchronous Execution Flow

# 3.1. Introducing Our Program

- “create\_process.c” in the text

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int main(int argc, char *argv[])
{
    int fd;
    int pid;
```

```
    printf("PID=%d) I am forking\n", getpid());
```

```
    pid = fork();
```

```
    if (pid == 0)
```

```
    {
```

```
        printf("PID=%d) I am running 'ls'\n", getpid());
```

```
        execl("/bin/ls", "", NULL);
```

```
        exit(2);
```

```
    }
```

```
    if(waitpid(pid, NULL, 0) < 0)
```

```
        printf("PID=%d) wait error (waiting PID %d)\n", getpid(), pid);
```

```
    else
```

```
        printf("PID=%d) wait succeeded (waiting PID %d)\n", getpid(), pid);
```

```
    printf("PID=%d) I am forking\n", getpid());
```

```
    pid = fork();
```

```
    if (pid == 0){
```

```
        printf("PID=%d) I am opening a text file\n", getpid());
```

```
        fd=open("Chapter_03.txt", O_RDONLY);
```

```
        close(fd);
```

```
    }
```

```
    if(waitpid(pid, NULL, 0) < 0)
```

```
        printf("PID=%d) wait error (waiting PID %d)\n", getpid(), pid);
```

```
    else
```

```
        printf("PID=%d) wait succeeded (waiting PID %d)\n", getpid(), pid);
```

```
    exit(0);
```

PID=30051) I am forking

PID=30052) I am running 'ls'

Chapter\_03.txt count.c create\_process.c foorig.c

a.out count.s foo.c testlist.c

PID=30051) wait succeeded (waiting PID 30052)

PID=30051) I am forking

PID=30053) I am opening a text file

PID=30053) wait error (waiting PID 0)

PID=30051) wait succeeded (waiting PID 30053)



# 3.1. Introducing Our Program

- Process
- Thread
- Kernel Thread

# 3.1. Introducing Our Program

- Process

(system call)

- fork

- 

- (

- )

- exec

- 

- exit

- 

- wait

- 

- (

- 가 exit 가  
processor descriptor

- )

# 3.1. Introducing Our Program

- Process

```
x = 10000;
y = fork();
if (!y) {
    x += 10000;
    printf("I am the child, x=%d\n", x);
} else {
    sleep(5);
    x -= 10000;
    printf("I am the parent, x=%d\n", x);
    wait(&status);
}
```

# 3.1. Introducing Our Program

- Process

- Thread

- pthread\_create(..., func, ...)

```
__clone(func, ..., clone_flags, ...) {  
    .....  
    y = clone(clone_flags, ...);  
    if (x == 0) {  
        func(...);  
    }  
    .....  
}
```

- clone                      fork                      ,                      do\_fork                      가

# 3.1. Introducing Our Program

- Process
- Thread
- Kernel Thread

```
int kernel_thread(int (*fn) (void *), void* arg, long flags)
```

```
(    do_fork    가    )
```

# 3.1. Introducing Our Program

- System call
  - `sys_call_table[]` in `arch/i386/kernel/syscall_table.S`
  - `sys_***()`
  - `do_***()` (sometimes)

# Contents

3.0. Intro.

3.1. Introducing Our Program

**3.2. Process Descriptor**

3.3. Process Creation: `fork()`, `vfork()`, & `clone()`

3.4. Process Lifespan

3.5. Process Termination

3.6. Keeping Track of Process: Basic Scheduler  
Construction

3.7. Wait Queues

3.8. Asynchronous Execution Flow

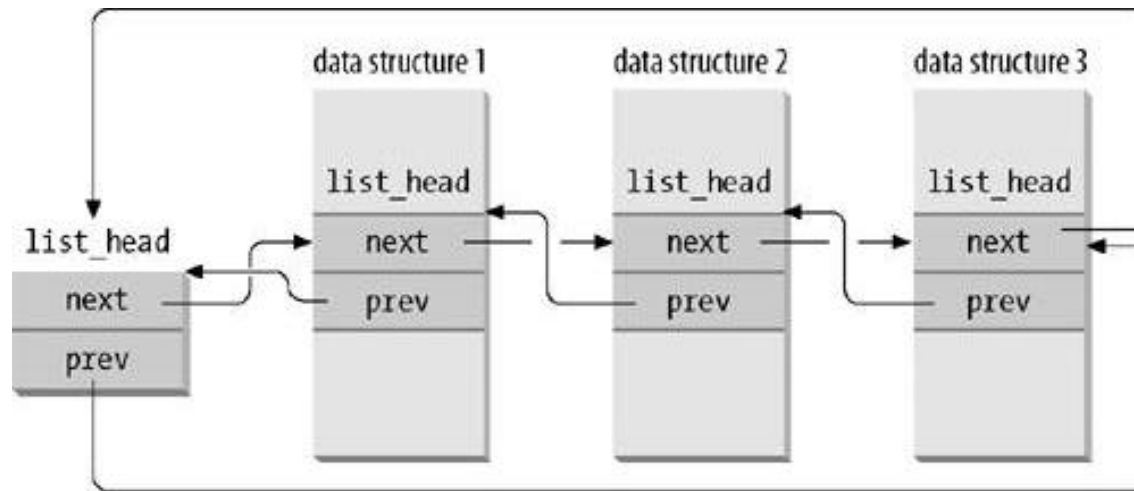
## 3.2. Process Descriptor

- Process Descriptor (PCB)
  - : struct task\_struct
- Allocated by
  - : alloc\_task\_struct()  
(whenever a process is created)
- “current”
  - : points the Process descriptor of current process  
 (“current->\*\*\*” is frequently used).



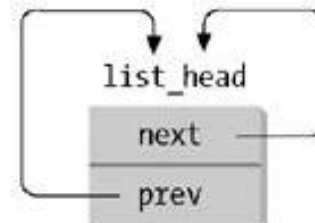
## 3.2. Process Descriptor

- Doubly Linked List



(a) a doubly linked list with three elements

(b) an empty doubly linked list



## 3.2. Process Descriptor

- Doubly Linked List

```
#define for_each_process(p) \  
    for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

- Where ***init\_task***: the proc. desc. of “init” (pid 1)  
( )

## 3.2. Process Descriptor

- `thread_info`
    - contains low level info. about the process.
    - *entry.S* accesses them immediately.
    - should fit inside of 1 cache line.
  - Allocated by
    - `alloc_thread_info()`
    - declaring a ***thread\_union*** variable
- \* always 2pages = 8kB (... or 4kB)

## 3.2. Process Descriptor

- `thread_info`

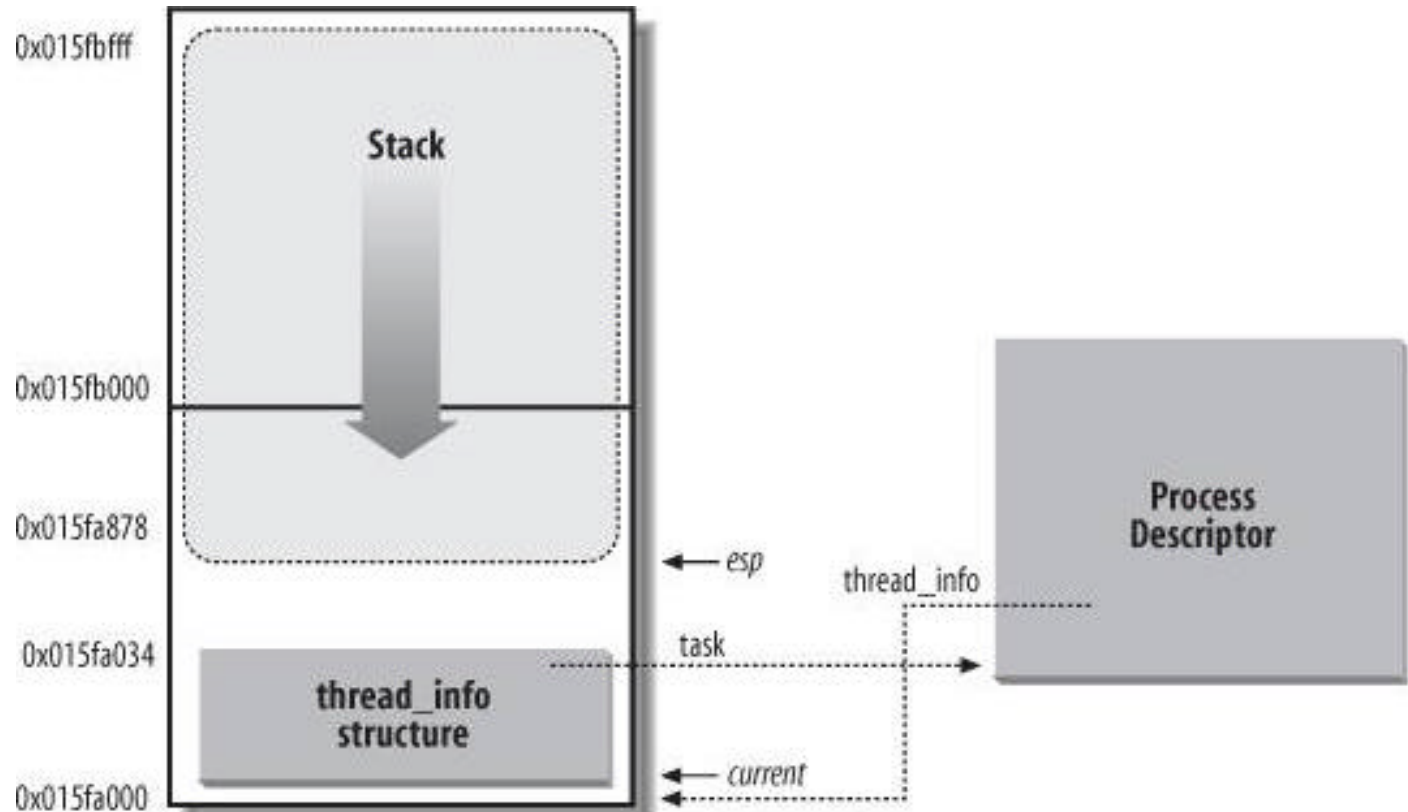
- shares the Kernel Mode Stack pages

(2 thread\_info 가  
,  
)

- context(registers) saving
    - stack frame for function call

## 3.2. Process Descriptor

- `thread_info`



## 3.2. Process Descriptor

- thread\_info

- how “current” is implemented

- current\_thread\_info()

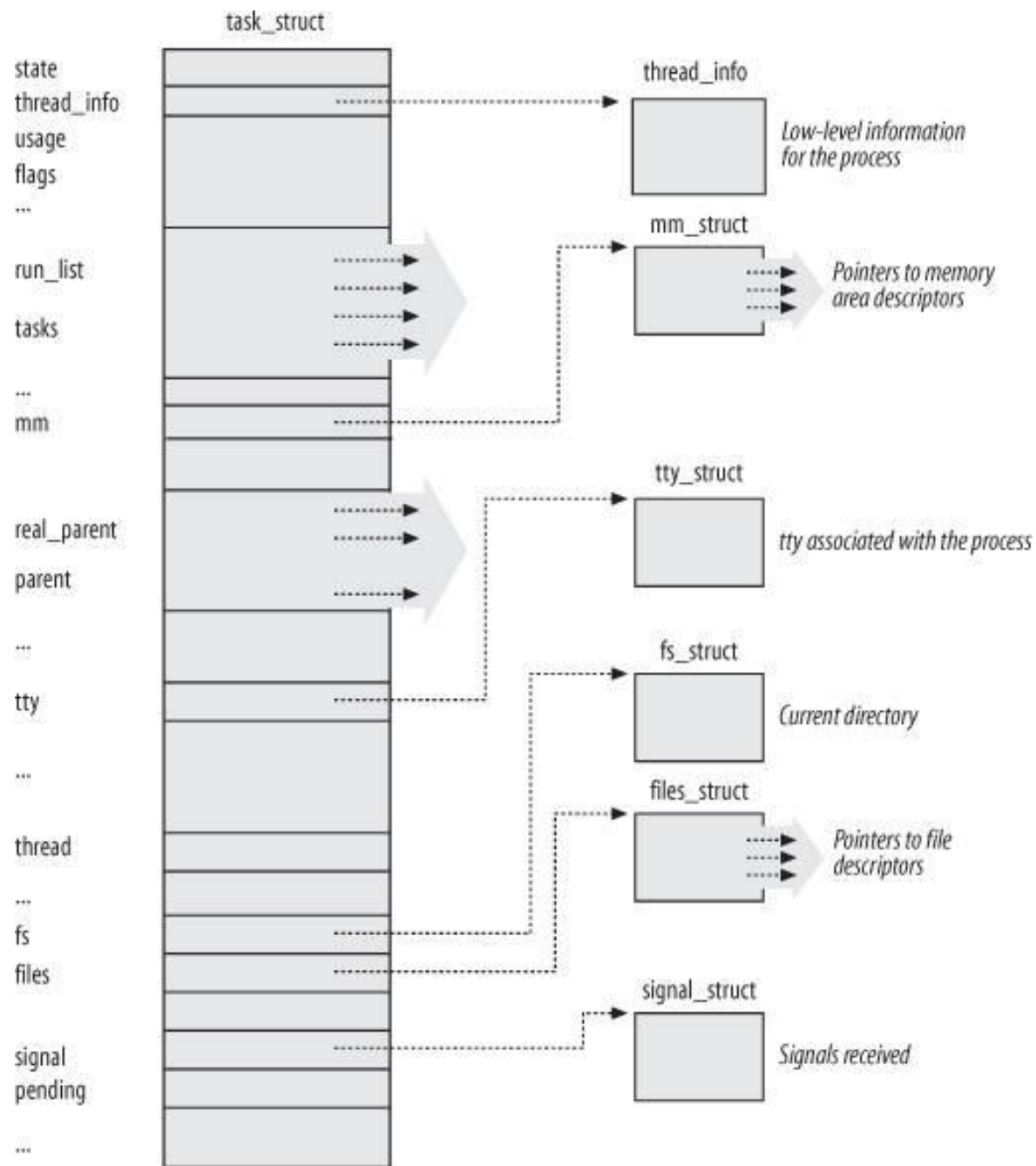
- movl \$0xffffe000,%ecx /\* or 0xfffff000 for 4KB stacks \*/
      - andl %esp,%ecx
      - movl %ecx,p

- current (= current\_thread\_info()->task )

- movl \$0xffffe000,%ecx /\* or 0xfffff000 for 4KB stacks \*/
      - andl %esp,%ecx
      - movl (%ecx),p

## 3.2. Process Descriptor

- Fields of task\_struct (1/4)





## 3.2. Process Descriptor

- Fields of task\_struct (2/4)

- about scheduling

- prio

- dynamic priority

- depends on scheduling history & static\_prio

- in range of static\_prio  $\pm 5$

- static\_prio

- can be set by nice() system call

- set\_user\_nice() modifies static\_prio

- default: MAX\_PRIO - 20

```
#define NICE_TO_PRIO(nice) (MAX_RT_PRIO + (nice) + 20)
```

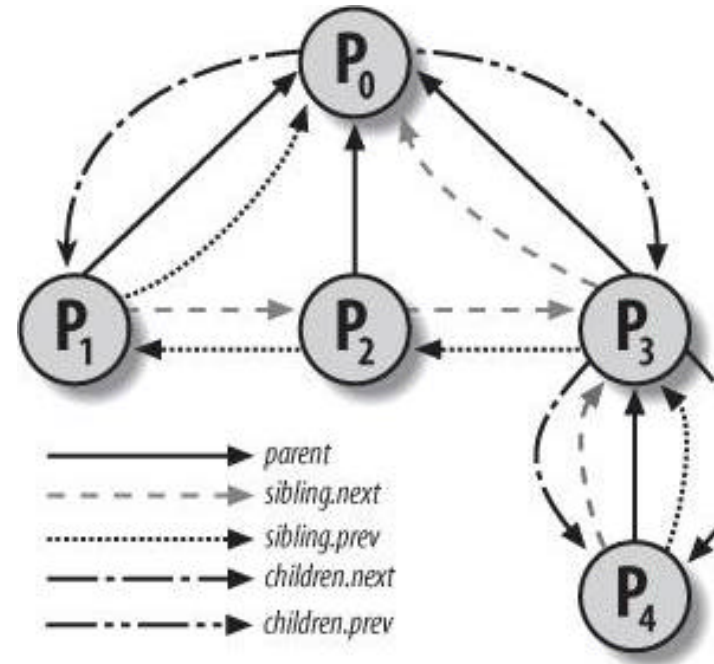
```
#define PRIO_TO_NICE(prio) ((prio) - MAX_RT_PRIO - 20)
```

```
(-20 <= nice <= 19)
```

## 3.2. Process Descriptor

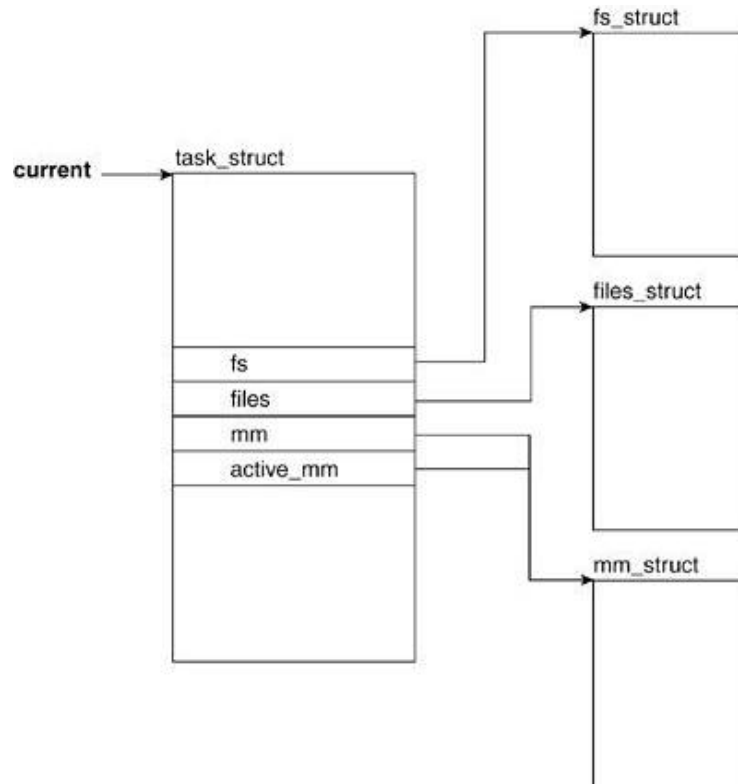
### Fields of task\_struct (3/4)

- about relationships
  - real\_parent
  - parent
    - can point proc.desc. of ptrace
  - sibling
    - used to be a node of a linked list
  - children
    - list head of the children's sibling list



## 3.2. Process Descriptor

- Fields of task\_struct (4/4)
  - about file system & address space



# Contents

3.0. Intro.

3.1. Introducing Our Program

3.2. Process Descriptor

**3.3. Process Creation: fork(), vfork(), & clone()**

3.4. Process Lifespan

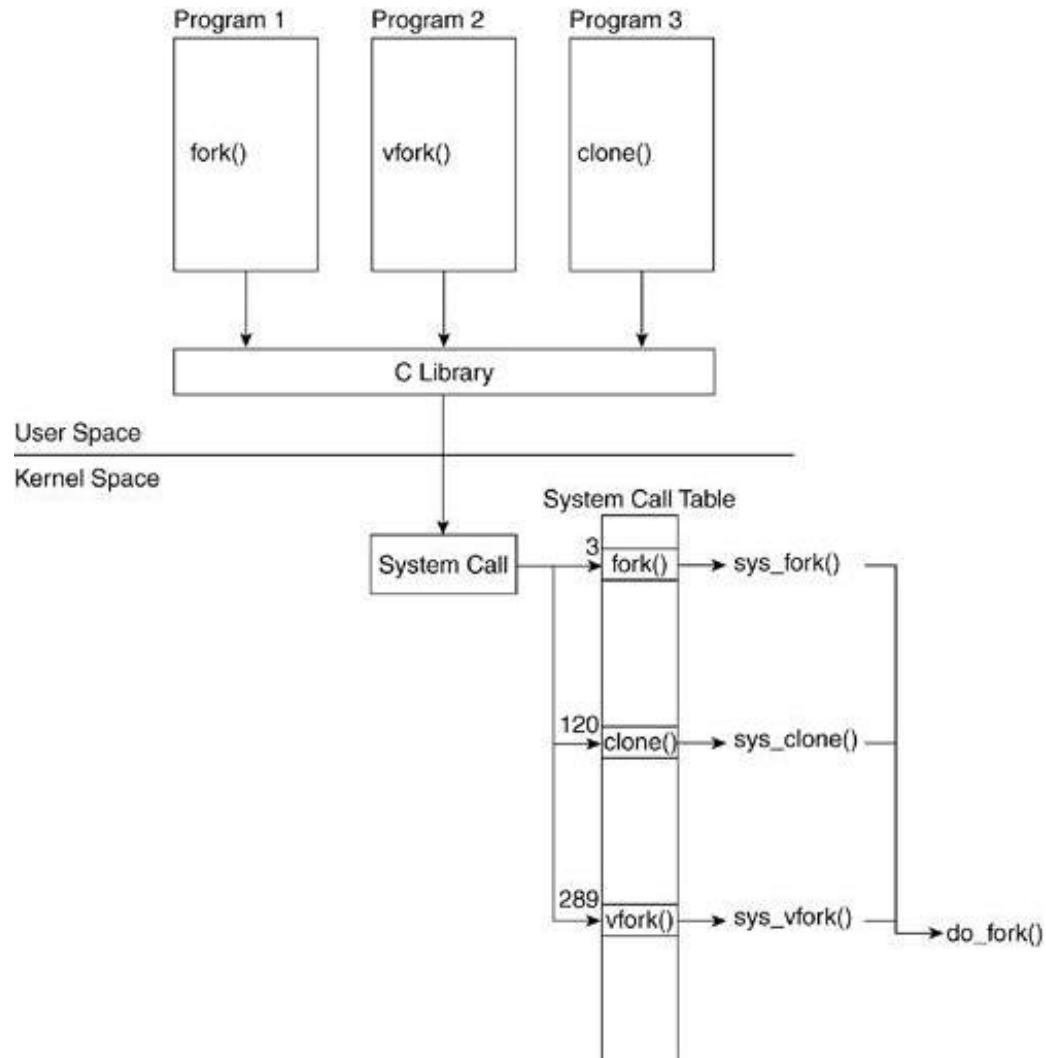
3.5. Process Termination

3.6. Keeping Track of Process: Basic Scheduler  
Construction

3.7. Wait Queues

3.8. Asynchronous Execution Flow

## 3.3. Process Creation



## 3.3. Process Creation

- `man clone`
  - usually used for implement 'thread'
  - take arguments 'fn' & 'args'
  - can share some resource with the parent
  - not POSIX

## 3.3. Process Creation

- Flags passed to `do_fork()`

	<code>fork()</code>	<code>vfork()</code>	<code>clone()</code>
<code>SIGCHLD</code>	V	V	
<code>CLONE_VFORK</code>		V	
<code>CLONE_VM</code>		V	

## 3.3. Process Creation

- Let's investigate
  - `do_fork()`
  - `copy_process()`
    - generates process descriptor & instances of many other data structures



## 3.3. Process Creation

- `copy_process()`가 (1/2)
  - `p = dup_task_struct();`
    - `child`      `task_struct`, `thread_info`      kernel mode stack
    - “`alloc_***_()`” :
    - “`*ti = *orig->thread_info; *tsk = *orig;`” :
  - `proc. desc.`
    - `pid`, `children`, `sibling`, `sigpending`, `start_time`
  - Resource
    - `clone_flags`      `CLONE_VM`, `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND` bit field ,  
fs, mm, file, sig-handler
    - `copy_files()`, `copy_fs()`, `copy_sighand()`, `copy_mm()`

# 3.3. Process Creation

- `copy_process()`가 (2/2)
  - Kernel mode stack (retval = `copy_thread(...);`)
    - , `eax` ( fork return )
    - , stack pointer (`esp`) return address (`eip`)
  - \* `ret_from_fork` *arch/i386/kernel/entry.S*
  - (sched\_fork(p);)
  - proc. desc. linked list (SET\_LINKS(p);)
  - do\_fork() 가 , RUNNING
  - run queue (wake\_up\_forked\_process(p);)
  - (if (...) set\_need\_resched();)

# Contents

3.0. Intro.

3.1. Introducing Our Program

3.2. Process Descriptor

3.3. Process Creation: `fork()`, `vfork()`, & `clone()`

**3.4. Process Lifespan**

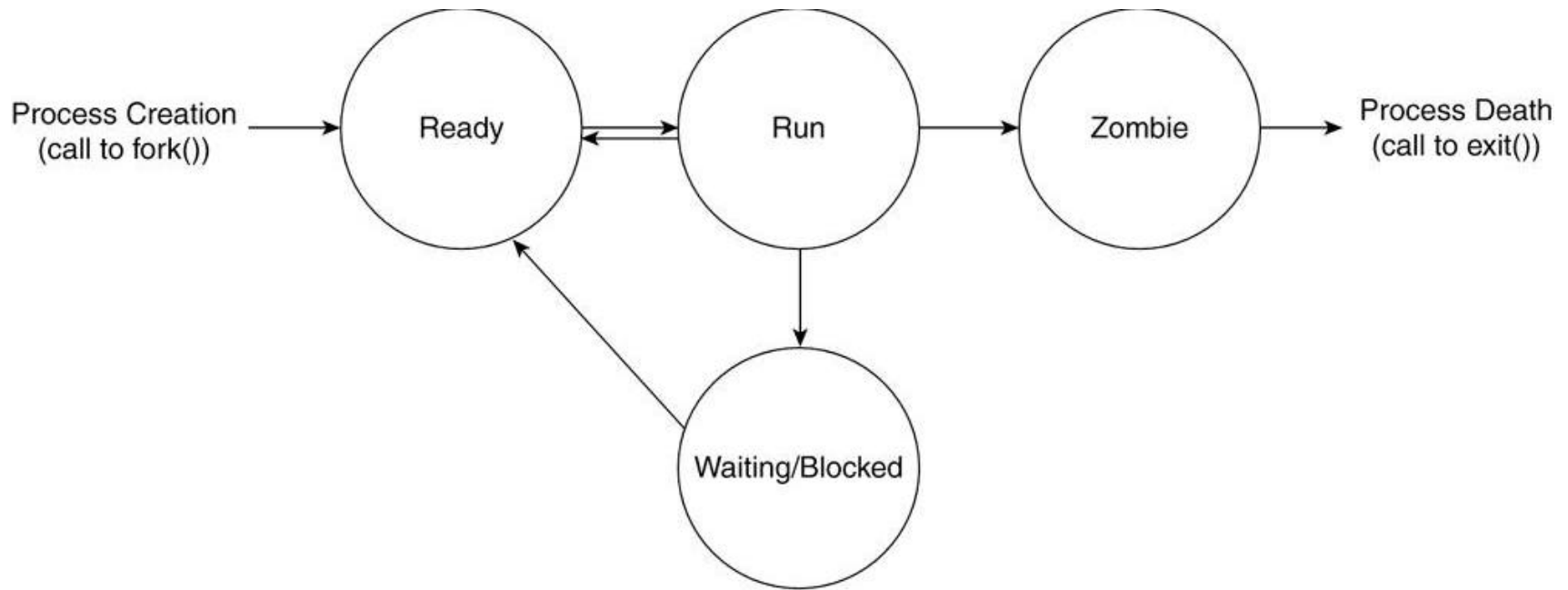
3.5. Process Termination

3.6. Keeping Track of Process: Basic Scheduler Construction

3.7. Wait Queues

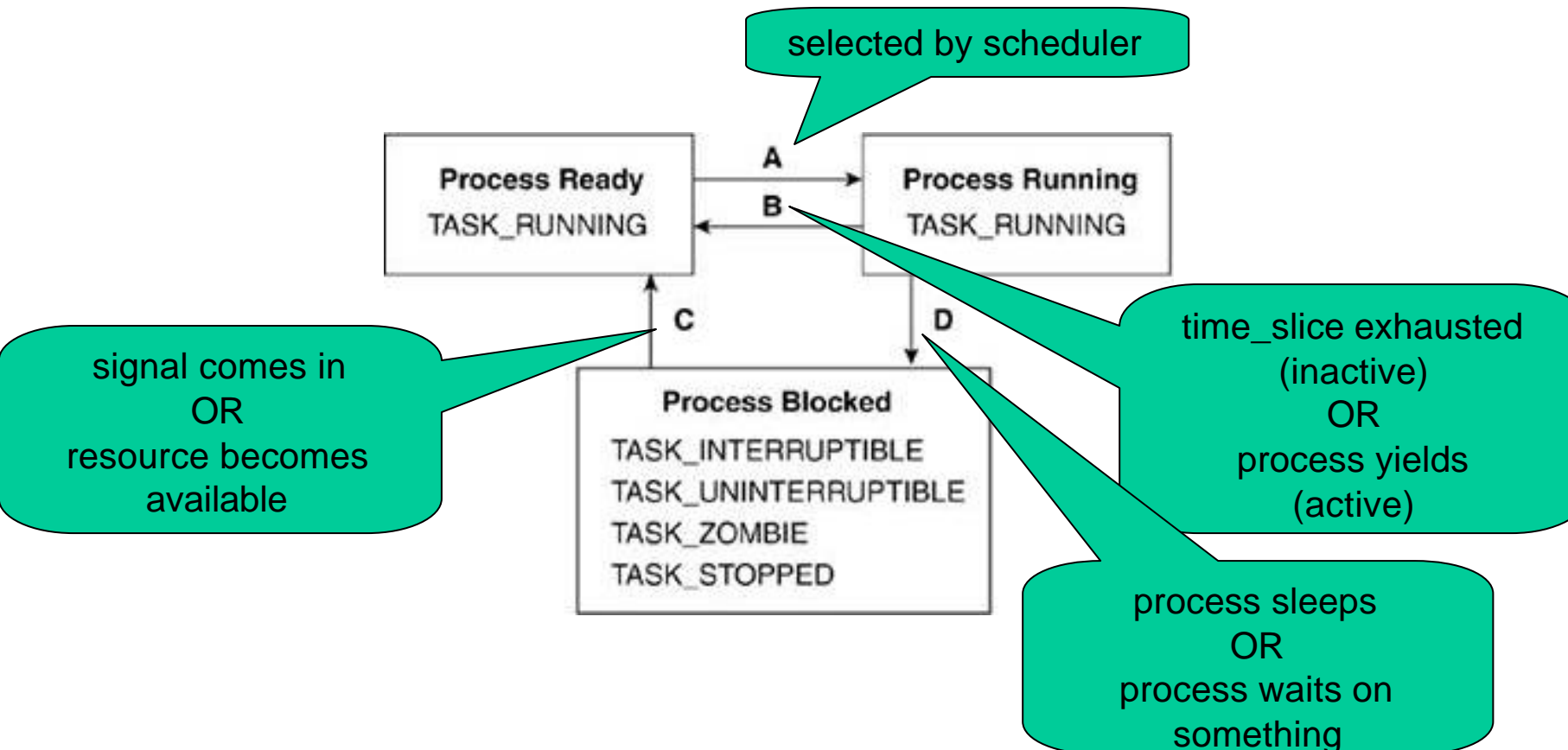
3.8. Asynchronous Execution Flow

## 3.4. Process Lifespan



## 3.4. Process Lifespan

- transition of “state” field in task\_struct



## 3.4. Process Lifespan

- TASK\_RUNNING to TASK\_INTERRUPTIBLE
  - event resource blocking I/O 가
  - resource가 signal , wake up .
  - 가 .

```
while (1) {  
    if (resource_available) break;  
    set_current_state(TASK_INTERRUPTIBLE);  
    schedule();  
}  
set_current_state(TASK_RUNNING);
```
  - interruptible\_sleep\_on()

## 3.4. Process Lifespan

- TASK\_RUNNING to TASK\_UNINTERRUPTIBLE
  - TASK\_INTERRUPTIBLE, signal
  - do\_fork()  
state default
  - device probing interrupt
  - TASK\_UNINTERRUPTIBLE  
sleep\_on()

## 3.4. Process Lifespan

- TASK\_RUNNING to TASK\_ZOMBIE

- exit 가 wait .
- resource proc. desc. .
- sys\_exit() TASK\_ZOMBIE 가 .
- 가 wait , ( 가 kill ).

- TASK\_RUNNING to TASK\_STOPPED

- 1: trace
- 2: SIGSTOP



# Contents

3.0. Intro.

3.1. Introducing Out Program

3.2. Process Descriptor

3.3. Process Creation: `fork()`, `vfork()`, & `clone()`

3.4. Process Lifespan

**3.5. Process Termination**

3.6. Keeping Track of Process: Basic Scheduler Construction

3.7. Wait Queues

3.8. Asynchronous Execution Flow

## 3.5. Process Termination

- When does a process terminate?
  - calling `exit()` explicitly
  - returning from `main()`  
(internally calling `exit()`)

## 3.5. Process Termination

- `sys_exit()`
  - just converts the format of `error_code`
  - calls `do_exit()`

# 3.5. Process Termination

- `do_exit()` 가 (1/2)
  - `exit` ( )  
`current->flags |= PF_EXITING;`
  - core image  
`__exit_mm(current);`
  - resource  
`exit_sem(current); __exit_files(current); __exit_fs(current);`  
`exit_namespace(current); exit_thread();`
  - `if (current->signal->leader) disassociate_ctty(1);`
  - `module_put();`

## 3.5. Process Termination

- `do_exit()` 가 (2/2)
  - exit code     `current->exit_code`
  - `exit_notify();` //
    - `current`     `child_reaper(=init)`
    - `forget_original_parent(current);`
    - `do_notify_parent(...);`
      - `signal`     `(__group_send_sig_info())`
      - `(__wake_up_parent())`
    - `current`     ZOMBIE
      - `current->state = TASK_ZOMBIE;`
  - Reschedule
    - `schedule();`

## 3.5. Process Termination

- 가 : wait
  - Mortician: getting task death information
  - Grave digger: getting rid of all traces of a process

## 3.5. Process Termination

- `sys_wait4()`가 (1/2)
  - wait queue
    - `DECLARE_WAITQUEUE(wait, current);`  
`add_wait_queue(&current->wait_chldexit, &wait);`
  - blocked
    - `current->state = TASK_INTERRUPTIBLE;`

# 3.5. Process Termination

- `sys_wait4()` 71 (2/2)
  - list\_for\_each(\_p, &tsk->children) {  
    p = list\_entry(\_p, struct task\_struct, sibling);  
    ...  
– wait\_task\_zombie()  
    release\_task()  
        proc\_pid\_unhash(); \_\_exit\_signal();  
        \_\_exit\_sighand(); \_\_unhash\_process();  
        sched\_exit(); release\_thread();  
        put\_task\_struct();



# Contents

3.0. Intro.

3.1. Introducing Our Program

3.2. Process Descriptor

3.3. Process Creation: `fork()`, `vfork()`, & `clone()`

3.4. Process Lifespan

3.5. Process Termination

**3.6. Keeping Track of Process: Basic  
Scheduler Construction**

3.7. Wait Queues

3.8. Asynchronous Execution Flow

## 3.6. Keeping Track of Processes

- runqueue (1 per CPU)

# struct runqueue

prio\_array\_t \*active

spinlock\_t lock

task\_t \*curr, \*idle

prio\_array\_t \*expired

unsigned long nr\_running

init best\_expired\_prio

prio\_array\_t arrays[0]

int nr\_active

unsigned long bitmap [BITMAP\_SIZE]

struct list\_head queue [MAX\_PRIO]

0

1

2

3

4

5

6

138

139

140

X

Y

...

Z

(queue of priority 138 task pointers)

A

B

...

C

(queue of priority 0 task pointers)

prio\_array\_t arrays[1]

int nr\_active

unsigned long bitmap [BITMAP\_SIZE]

struct list\_head queue [MAX\_PRIO]

0

1

2

3

4

5

6

138

139

140

U

V

...

W

(queue of priority 138 task pointers)

D

E

...

F

(queue of priority 0 task pointers)

?

