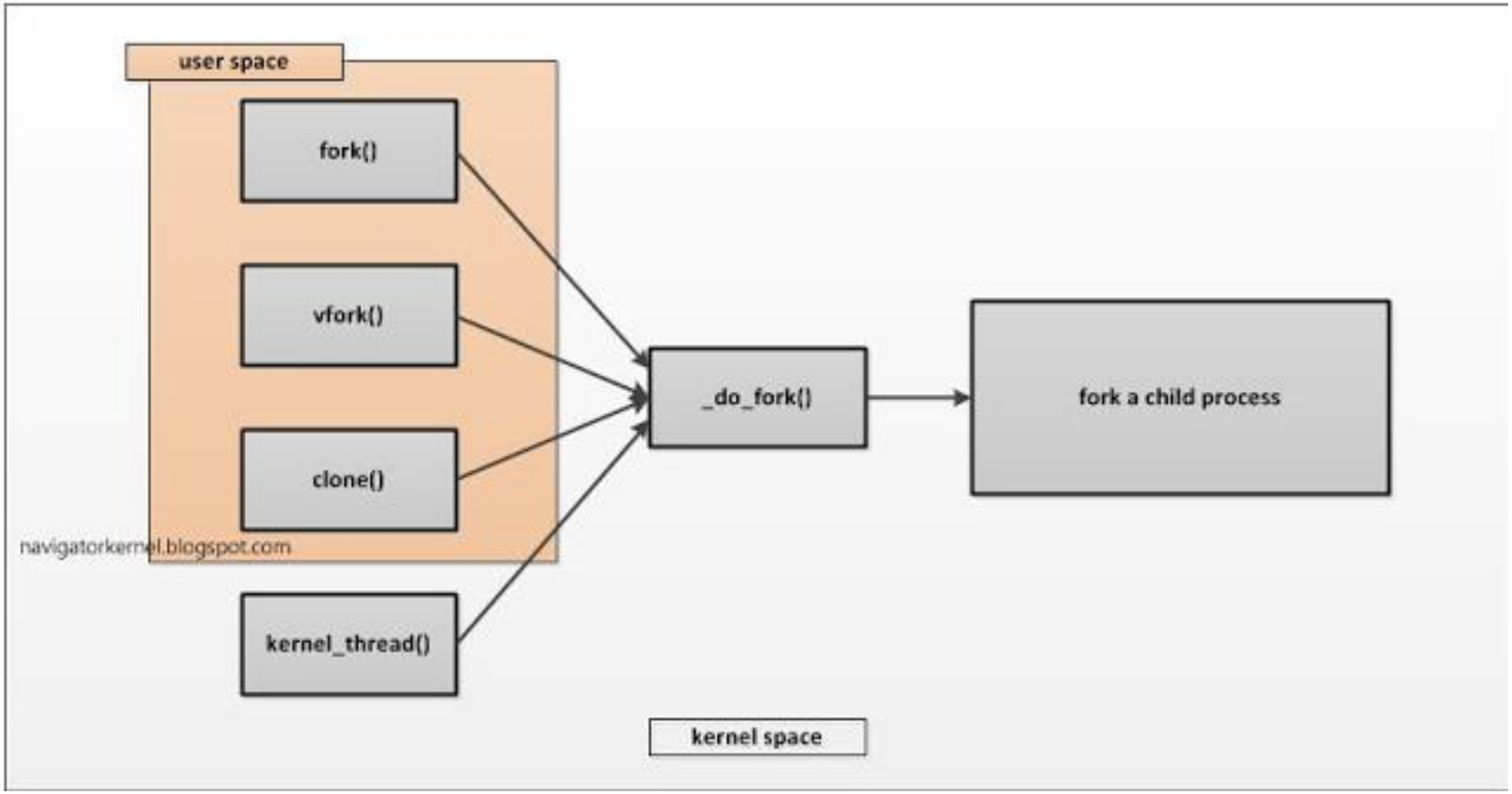
Process Management and Scheduling

2.4 Process Management System Calls



2.4 Process Management System Calls

```c
int sys_fork(struct pt_regs *regs)
{
        return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}

/*
 * This is trivial, and on the face of it looks like it
 * could equally well be done in user mode.
 *
 * Not so, for quite unobvious reasons - register pressure.
 * In user mode vfork() cannot have a stack frame, and if
 * done by calling the "clone()" system call directly, you
 * do not have enough call-clobbered registers to hold all
 * the information you need.
 */
int sys_vfork(struct pt_regs *regs)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, reg
                        NULL, NULL);
}

long
sys_clone(unsigned long clone_flags, unsigned long newsp,
          void __user *parent_tid, void __user *child_tid, s
{
        if (!newsp)
                newsp = regs->sp;
        return do_fork(clone_flags, newsp, regs, 0, parent_t
}
```

```c
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    return _do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn,
        (unsigned long)arg, NULL, NULL, 0);
}

#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
#else
    /* can not support in nommu mode */
    return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
SYSCALL_DEFINE0(vfork)
{
    return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
            0, NULL, NULL, 0);
}
#endif

#ifdef __ARCH_WANT_SYS_CLONE
#ifdef CONFIG_CLONE_BACKWARDS
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
         int __user *, parent_tidptr,
         unsigned long, tls,
         int __user *, child_tidptr)
```

Executing System Calls

```c
long do_fork(unsigned long clone_flags,
          unsigned long stack_start,
          unsigned long stack_size,
          int __user *parent_tidptr,
          int __user *child_tidptr)
{
    return _do_fork(clone_flags, stack_start, stack_size,
           parent_tidptr, child_tidptr, 0);
}
```

```c
long _do_fork(unsigned long clone_flags,
          unsigned long stack_start,
          unsigned long stack_size,
          int __user *parent_tidptr,
          int __user *child_tidptr,
          unsigned long tls)
{
    struct task_struct *p;
    int trace = 0;
    long nr;

    /*
     * Determine whether and which event to report to ptracer.  When
     * called from kernel_thread or CLONE_UNTRACED is explicitly
     * requested, no event is reported; otherwise, report if the event
     * for the type of forking is enabled.
     */
    if (!(clone_flags & CLONE_UNTRACED)) {
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK;
        else if ((clone_flags & CSIGNAL) != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;

        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }

    p = copy_process(clone_flags, stack_start, stack_size,
             child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
    add_latent_entropy();
```

Copy on Write

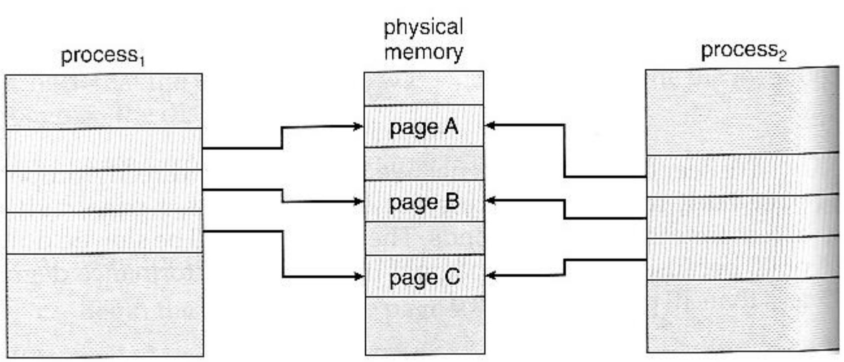


Figure 9.7 Before process 1 modifies page C.

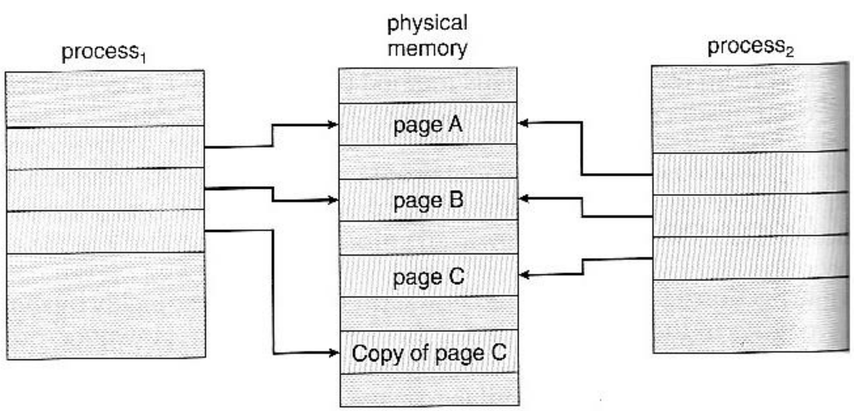


Figure 9.8 After process 1 modifies page C.

_do_fork

```
nr = task_pid_vnr(p);
```

```
pid = get_task_pid(p, PIDTYPE_PID);
nr = pid_vnr(pid);
```

```
pid_t __task_pid_nr_ns(struct task_struct *task, enum pid_type type,
                        struct pid_namespace *ns)
{
        pid_t nr = 0;

        rcu_read_lock();
        if (!ns)
                ns = current->nsproxy->pid_ns;
        if (likely(pid_alive(task))) {
                if (type != PIDTYPE_PID)
                        task = task->group_leader;
                nr = pid_nr_ns(task->pids[type].pid, ns);
        }
        rcu_read_unlock();

        return nr;
}
```

```
struct pid *get_task_pid(struct task_struct *task, enum pid_type type)
{
    struct pid *pid;
    rcu_read_lock();
    if (type != PIDTYPE_PID)
        task = task->group_leader;
    pid = get_pid(rcu_dereference(task->pids[type].pid));
    rcu_read_unlock();
    return pid;
}
```

```
pid_t pid_vnr(struct pid *pid)
{
    return pid_nr_ns(pid, task_active_pid_ns(current));
}
```

```
pid_t pid_nr_ns(struct pid *pid, struct pid_namespace *ns)
{
        struct upid *upid;
        pid_t nr = 0;

        if (pid && ns->level <= pid->level) {
                upid = &pid->numbers[ns->level];
                if (upid->ns == ns)
                        nr = upid->nr;
        }
        return nr;
}
```

```
pid_t pid_nr_ns(struct pid *pid, struct pid_namespace *ns)
{
    struct upid *upid;
    pid_t nr = 0;

    if (pid && ns->level <= pid->level) {
        upid = &pid->numbers[ns->level];
        if (upid->ns == ns)
            nr = upid->nr;
    }
    return nr;
}
```

copy_proces

프로세스 복사

```
p = dup_task_struct(current, node);
if (!p)
    goto fork_out;
```

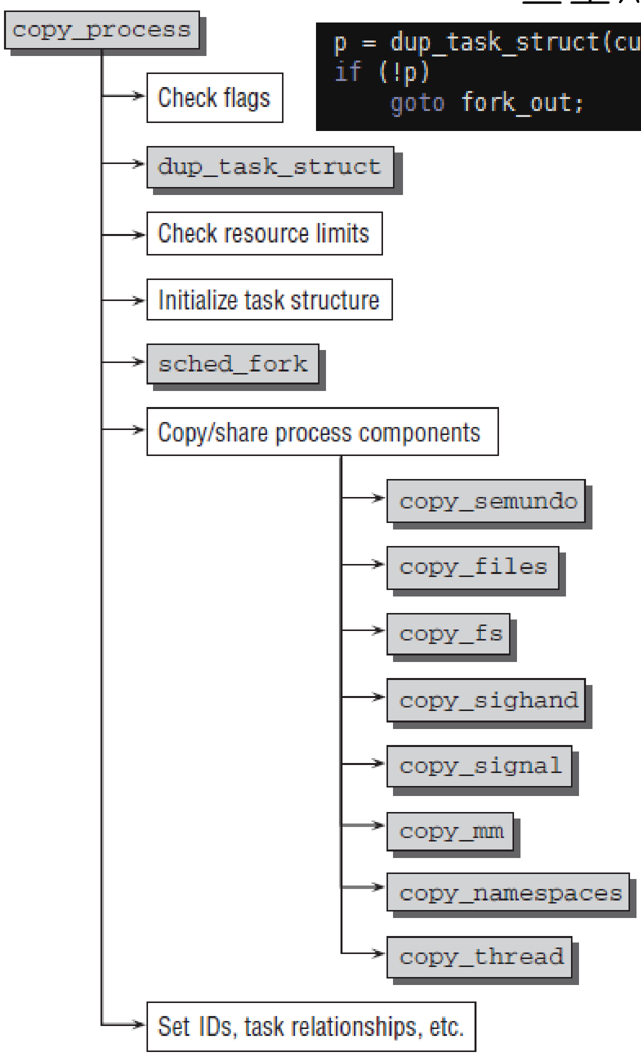


Figure 2-8: Code flow diagram for **copy_process**.

```
static __latent_entropy struct task_struct *copy_process(
					unsigned long clone_flags,
					unsigned long stack_start,
					unsigned long stack_size,
					int __user *child_tidptr,
					struct pid *pid,
					int trace,
					unsigned long tls,
					int node)
{
	int retval;
	struct task_struct *p;

	if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
		return ERR_PTR(-EINVAL);

	if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))
		return ERR_PTR(-EINVAL);

	/*
	 * Thread groups must share signals as well, and detached threads
	 * can only be started up within the thread group.
	 */
	if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
		return ERR_PTR(-EINVAL);

	/*
	 * Shared signal handlers imply shared VM. By way of the above,
	 * thread groups also imply shared VM. Blocking this case allows
	 * for various simplifications in other code.
	 */
	if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
		return ERR_PTR(-EINVAL);

	/*
	 * Siblings of global init remain as zombies on exit since they are
	 * not reaped by their parent (swapper). To solve this and to avoid
	 * multi-rooted process trees, prevent global and container-inits
	 * from creating siblings.
	 */
	if ((clone_flags & CLONE_PARENT) &&
				current->signal->flags & SIGNAL_UNKILLABLE)
		return ERR_PTR(-EINVAL);

	/*
	 * If the new process will be in a different pid or user namespace
	 * do not allow it to share a thread group with the forking task.
	 */
	if (clone_flags & CLONE_THREAD) {
		if ((clone_flags & (CLONE_NEWUSER | CLONE_NEWPID)) ||
		    (task_active_pid_ns(current) !=
				current->nsproxy->pid_ns_for_children))
			return ERR_PTR(-EINVAL);
```

flags 처리

thread_info

<asm-arch/thread_info.h>

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        unsigned long           flags;          /* low level flags */
        unsigned long           status;         /* thread-synchronous flags */
        __u32                   cpu;            /* current CPU */
        int                     preempt_count;  /* 0 => preemptable, <0 => BUG */

        mm_segment_t            addr_limit;     /* thread address space */
        struct restart_block    restart_block;
}
```

task – task structure pointer
flags – 2가지 중요 신호
TIF_SIGPENDING 보류 중 신호
TIF_NEED_RESCHED 프로세스가 스케줄러에 인해 대체
cpu – 프로세스가 실행되고 있는 cpu 번호
싱글 프로세스는 간단하지만 멀티 프로세스일 경우 중요
preemt_count – 커널 선점 카운트
addr_limit – 프로세스 주소 공간 지정

Kernel stack

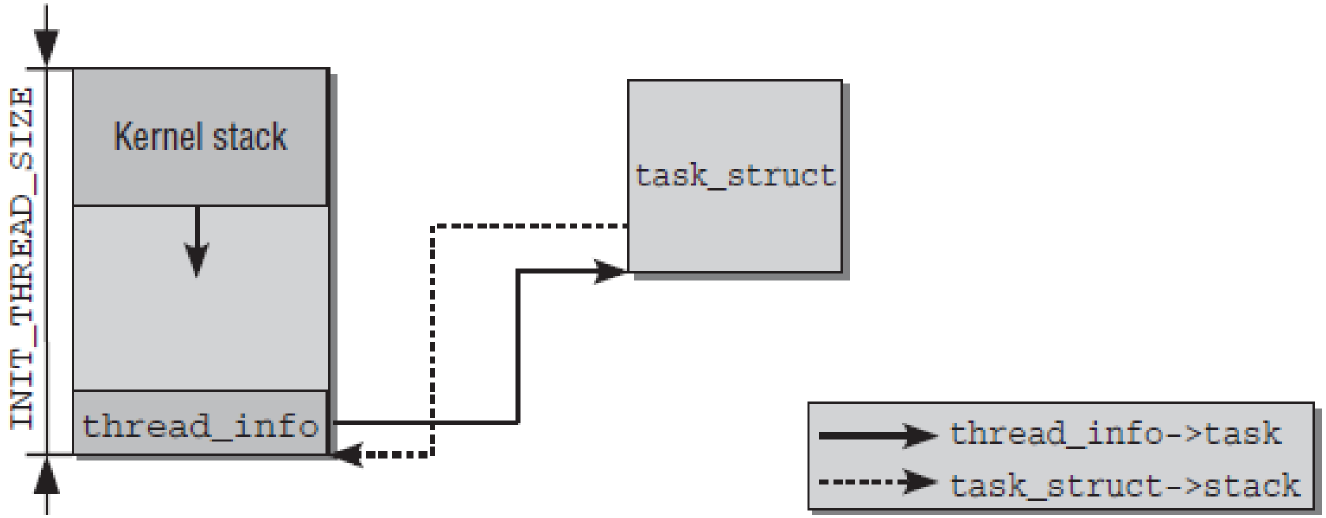


Figure 2-9: Relationship between **task_struct**, **thread_info**, and the kernel stack of a process.

```
#ifndef __HAVE_ARCH_KSTACK_END
static inline int kstack_end(void *addr)
{
    /* Reliable end of stack detection:
     * Some APM bios versions misalign the stack
     */
    return !(((unsigned long)addr+sizeof(void*)-1) & (THREAD_SIZE-sizeof(void*)));
}
#endif
```

process max number check

kernel/fork.c

```
if (atomic_read(&p->user->processes) >=
                p->signal->rlim[RLIMIT_NPROC].rlim_cur) {
        if (!capable(CAP_SYS_ADMIN) && !capable(CAP_SYS_RESOURCE) &&
            p->user != current->nsproxy->user_ns->root_user)
                goto bad_fork_free;
}
```

task_struct->real_cred->user_struct->processes

```
if (atomic_read(&p->real_cred->user->processes) >=
        task_rlimit(p, RLIMIT_NPROC)) {
    if (p->real_cred->user != INIT_USER &&
        !capable(CAP_SYS_RESOURCE) && !capable(CAP_SYS_ADMIN))
        goto bad_fork_free;
}
```

pid

```
if (clone_flags & CLONE_THREAD) {
        current->signal->nr_threads++;
        atomic_inc(&current->signal->live);
        atomic_inc(&current->signal->sigcnt);
        p->group_leader = current->group_leader;
        list_add_tail_rcu(&p->thread_group, &p->group_leader->thread_group);
}
```

전역 pid 계산

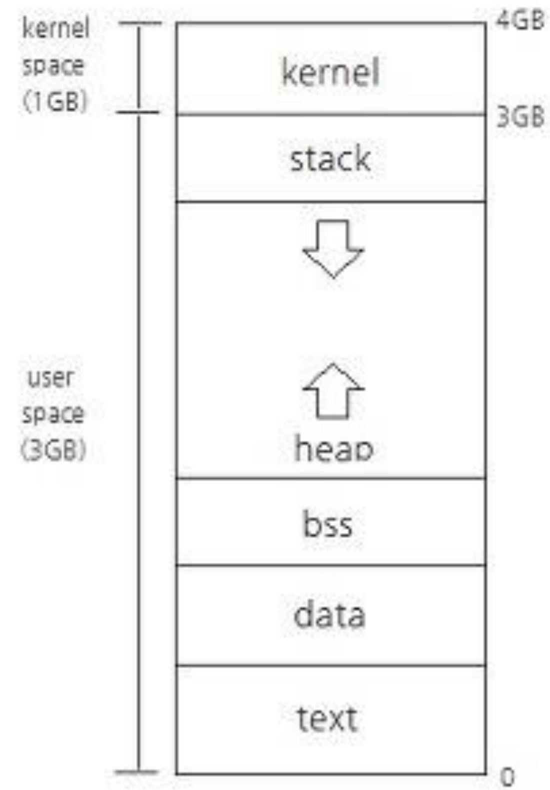
```
p->pid = pid_nr(pid);
p->tgid = p->pid;
if (clone_flags & CLONE_THREAD)
        p->tgid = current->tgid;
```

```
p->pid = pid_nr(pid);
if (clone_flags & CLONE_THREAD) {
    p->exit_signal = -1;
    p->group_leader = current->group_leader;
    p->tgid = current->tgid;
} else {
    if (clone_flags & CLONE_PARENT)
        p->exit_signal = current->group_leader->exit_signal;
    else
        p->exit_signal = (clone_flags & CSIGNAL);
    p->group_leader = p;
    p->tgid = p->pid;
}
```

```
if (likely(p->pid)) {
    ptrace_init_task(p, (clone_flags & CLONE_PTRACE) || trace);

    init_task_pid(p, PIDTYPE_PID, pid);
    if (thread_group_leader(p)) {
        init_task_pid(p, PIDTYPE_PGID, task_pgrp(current));
        init_task_pid(p, PIDTYPE_SID, task_session(current));

        if (is_child_reaper(pid)) {
            ns_of_pid(pid)->child_reaper = p;
            p->signal->flags |= SIGNAL_UNKILLABLE;
        }

        p->signal->leader_pid = pid;
        p->signal->tty = tty_kref_get(current->signal->tty);
        list_add_tail(&p->sibling, &p->real_parent->children);
        list_add_tail_rcu(&p->tasks, &init_task.tasks);
        attach_pid(p, PIDTYPE_PGID);
        attach_pid(p, PIDTYPE_SID);
        __this_cpu_inc(process_counts);
    } else {
        current->signal->nr_threads++;
        atomic_inc(&current->signal->live);
        atomic_inc(&current->signal->sigcnt);
        list_add_tail_rcu(&p->thread_group,
                          &p->group_leader->thread_group);
        list_add_tail_rcu(&p->thread_node,
                          &p->signal->thread_head);
    }
    attach_pid(p, PIDTYPE_PID);
    nr_threads++;
}
```

Kernel threads

- 블록 디바이스와 수정된 메모리 페이지의 동기화
- 스왑 영역에 메모리 페이지 작성
- 지연된 작업 관리
- 파일시스템 트랜잭션 저널 구현



Two type

1 – 특정 작업을 수행하기 위해 대기
2 – 주기적인 간격으로 실행, 사용률이 초과하거나 설정된 한계 값 이하로 떨어지는 것을 주기적으로 모니터링

exec

/kernel/exec.c ⟶ /fs/exec.c

do_execve() ⟶

```
SYSCALL_DEFINE3(execve,
        const char __user *, filename,
        const char __user *const __user *, argv,
        const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

```
int do_execve(struct filename *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
    struct user_arg_ptr argv = { .ptr.native = __argv };
    struct user_arg_ptr envp = { .ptr.native = __envp };
    return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
}
```

```
/*
 * sys_execve() executes a new program.
 */
static int do_execveat_common(int fd, struct filename *filename,
                  struct user_arg_ptr argv,
                  struct user_arg_ptr envp,
                  int flags)
{
```

exec

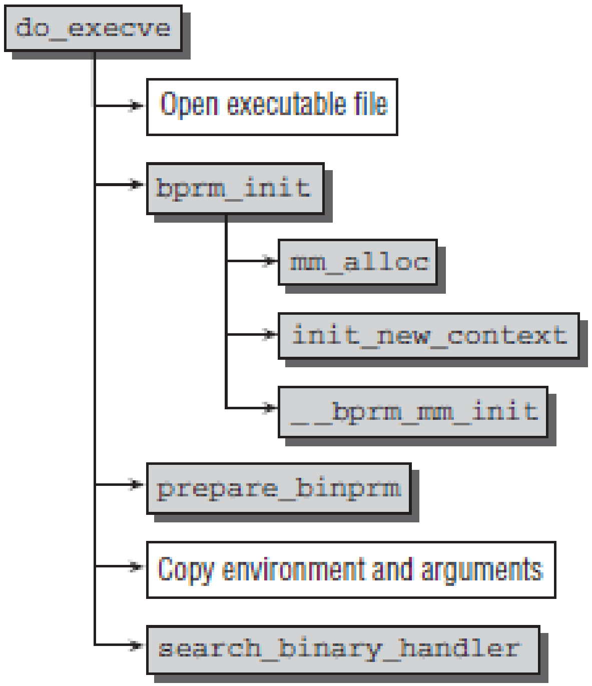


Figure 2-11: Code flow diagram for do_execve.

```
retval = bprm_mm_init(bprm);
if (retval)
        goto out_file;

bprm->argc = count(argv, MAX_ARG_STRINGS);
if ((retval = bprm->argc) < 0)
        goto out;

bprm->envc = count(envp, MAX_ARG_STRINGS);
if ((retval = bprm->envc) < 0)
        goto out;

retval = prepare_binprm(bprm);
if (retval < 0)
        goto out;
```

```
int bprm_mm_init(struct linux_binprm *bprm)
{
        int err;
        struct mm_struct *mm = NULL;

        bprm->mm = mm = mm_alloc();
        err = -ENOMEM;
        if (!mm)
                goto err;

        err = init_new_context(current, mm);
        if (err)
                goto err;

        err = __bprm_mm_init(bprm);
        if (err)
                goto err;

        return 0;

err:
        if (mm) {
                bprm->mm = NULL;
                mmdrop(mm);
        }

        return err;
}
```

```
static int bprm_mm_init(struct linux_binprm *bprm)
{
    int err;
    struct mm_struct *mm = NULL;

    bprm->mm = mm = mm_alloc();
    err = -ENOMEM;
    if (!mm)
        goto err;

    err = __bprm_mm_init(bprm);
    if (err)
        goto err;

    return 0;

err:
    if (mm) {
        bprm->mm = NULL;
        mmdrop(mm);
    }

    return err;
}
```

exec

```c
int prepare_binprm(struct linux_binprm *bprm)
{
	umode_t mode;
	struct inode * inode = bprm->file->f_path.dentry->d_inode;
	int retval;

	mode = inode->i_mode;
	if (bprm->file->f_op == NULL)
		return -EACCES;

	/* clear any previous set[ug]id data from a previous binary */
	bprm->cred->euid = current_euid();
	bprm->cred->egid = current_egid();

	if (!(bprm->file->f_path.mnt->mnt_flags & MNT_NOSUID)) {
		/* Set-uid? */
		if (mode & S_ISUID) {
			bprm->per_clear |= PER_CLEAR_ON_SETID;
			bprm->cred->euid = inode->i_uid;
		}

		/* Set-gid? */
		/*
		 * If setgid is set but no group execute bit then this
		 * is a candidate for mandatory locking, not a setgid
		 * executable.
		 */
		if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)) {
			bprm->per_clear |= PER_CLEAR_ON_SETID;
			bprm->cred->egid = inode->i_gid;
		}
	}
	/* fill in binprm security blob */
	retval = security_bprm_set_creds(bprm);
	if (retval)
		return retval;
	bprm->cred_prepared = 1;

	memset(bprm->buf, 0, BINPRM_BUF_SIZE);
	return kernel_read(bprm->file, 0, bprm->buf, BINPRM_BUF_SIZE);
}
```

bprm_fill_uid()

```c
int prepare_binprm(struct linux_binprm *bprm)
{
	int retval;

	bprm_fill_uid(bprm);

	/* fill in binprm security blob */
	retval = security_bprm_set_creds(bprm);
	if (retval)
		return retval;
	bprm->cred_prepared = 1;

	memset(bprm->buf, 0, BINPRM_BUF_SIZE);
	return kernel_read(bprm->file, 0, bprm->buf, BINPRM_BUF_SIZE);
}
```

scheduler

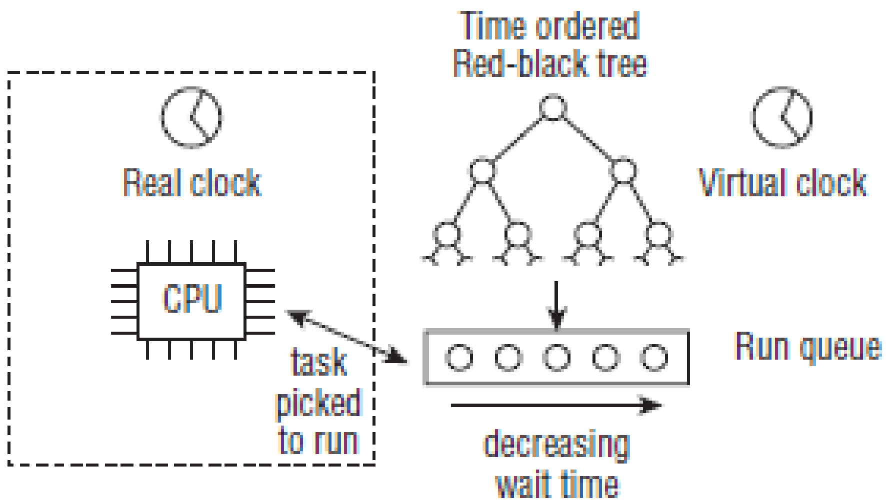


Figure 2-12: The scheduler keeps track of the waiting time of the available processes by sorting them in a red-black tree.

- Red−black tree 정렬
- 제일 왼쪽이 우선순위가 높음
- 왼쪽에서 오른쪽으로 정렬

- Real clock 보다 Virtual clock이 느리다
- Virtual clock은 대기 프로세스수에 따라 다르다.
  예) 4개면 1/4 real clock
- 실제 20초 = 4개의 프로세스 5초씩

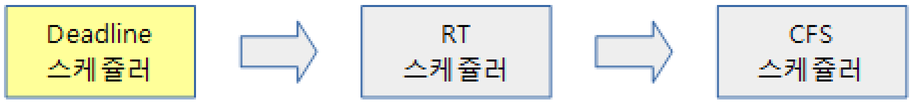
scheduler

generic scheduler

- 스케줄링 클래스는 다음에 실행될 태스크를 결정
- 여러 스케줄러 지원
- 태스크가 선정되면 CPU와 task switch가 발생

```
#define SCHED_NORMAL        0
#define SCHED_FIFO          1
#define SCHED_RR            2
#define SCHED_BATCH         3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE          5
```

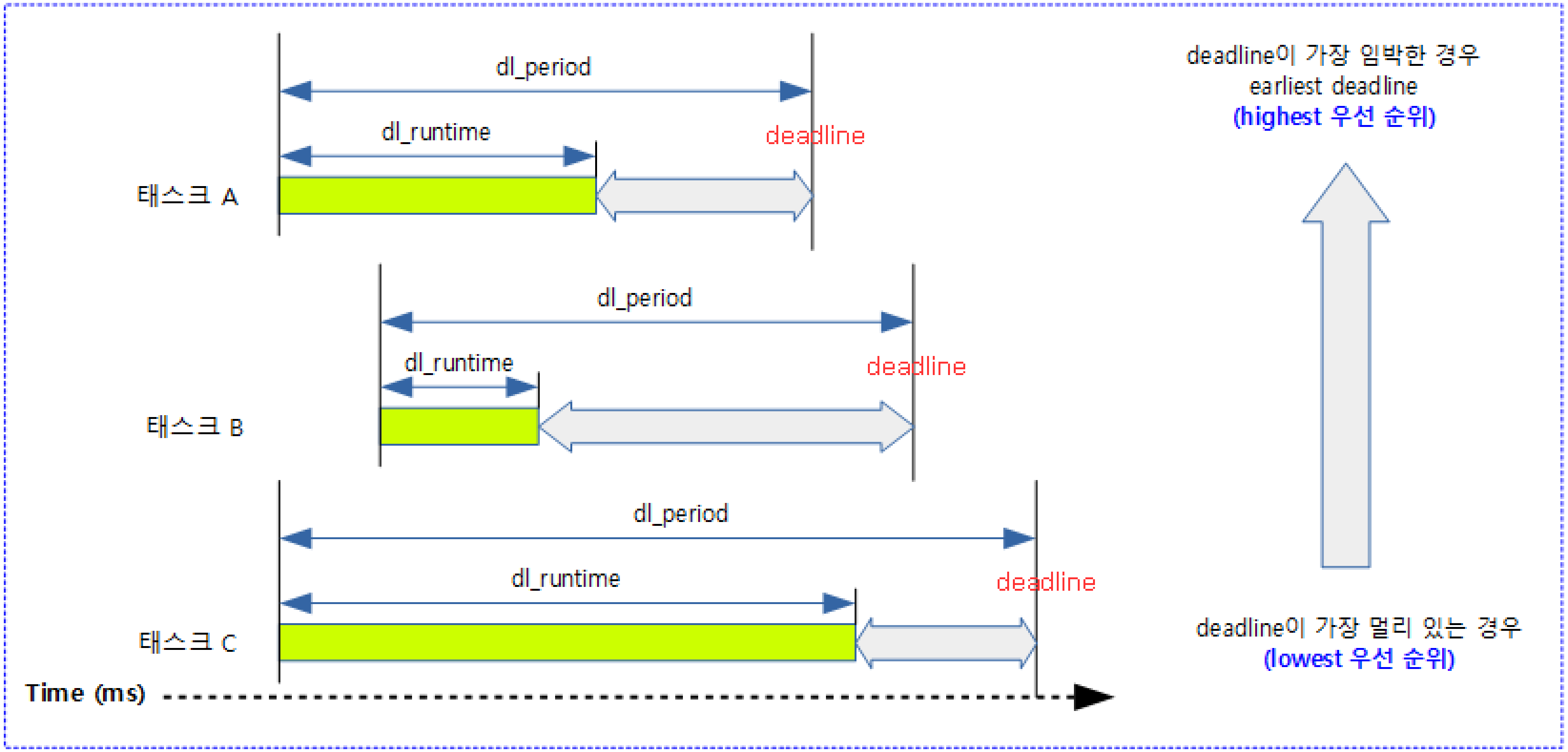
```
#define SCHED_NORMAL      0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
#define SCHED_DEADLINE    6
```



SCHED_NORMAL : CFS 동작
SCHED_BATCH : CFS, yield를 회피 하여 최대한 태스크 처리
SCHED_IDLE : CFS, 가장 낮은 우선순위로 동작
SCHED_FIFO : RT, FIFO 방식
SCHED_RR : RT, default 0.1초 단위로 round-robin 방식

scheduler

Deadline scheduler



Scheduler class

```
struct sched_class {
	const struct sched_class *next;

	void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
	void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
	void (*yield_task) (struct rq *rq);
	bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);

	void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);

	/*
	 * It is the responsibility of the pick_next_task() method that will
	 * return the next task to call put_prev_task() on the @prev task or
	 * something equivalent.
	 *
	 * May return RETRY_TASK when it finds a higher prio class has runnable
	 * tasks.
	 */
	struct task_struct * (*pick_next_task) (struct rq *rq,
						struct task_struct *prev,
						struct pin_cookie cookie);
	void (*put_prev_task) (struct rq *rq, struct task_struct *p);
```

- enqueue_task : 실행 대기열에 새 프로세스 추가, sleep -> run
- dequeue task : 실행 대기열에서 프로세스 가져옴, run -> un-run
- yield_task : 프로세스가 프로세서의 제어권 반납

scheduler

core scheduler

- Hz에 의해 자동 호출
- 리소스가 부족하면 자동 해제
- 수행되는 주요 작업 count 증가
- 주기적인 스케줄링 활성화

update_rq_clock()
runqueue time update

cpu_load_update_active()
cpu_load[] history array update

```
void scheduler_tick(void)
{
	int cpu = smp_processor_id();
	struct rq *rq = cpu_rq(cpu);
	struct task_struct *curr = rq->curr;

	sched_clock_tick();

	raw_spin_lock(&rq->lock);
	update_rq_clock(rq);
	curr->sched_class->task_tick(rq, curr, 0);
	cpu_load_update_active(rq);
	calc_global_load_tick(rq);
	raw_spin_unlock(&rq->lock);

	perf_event_task_tick();

#ifdef CONFIG_SMP
	rq->idle_balance = idle_cpu(cpu);
	trigger_load_balance(rq);
#endif
	rq_last_tick_reset(rq);
}
```

scheduler

sched_fork

fork(), clone()가 생성될 때 스케줄러가 sched_fork()를 사용해서 연결함

- 프로세스 스케줄링 초기화
- 데이터 구조 설정
- 동적 우선 순위 결정

context_switching

switch_mm : task_struct->mm에 설명된 메모리 전환

switch_to : 프로세서 레지스터 내용과 커널 스택 전환

Lazy FPU Mode

- context switching 속도가 시스템 성능에 큰 영향을 미침
- CPU 시간을 줄이기 위해 부동 소수점 레지스터는 실제 응용프로그램에서 사용하지 않으면 저장되지 않고 복구 되지 않는다.