Outline

1. Process Management
2. Context Switch
3. Summary
A process is a program in execution.

A process contains:
- Address space (e.g. read-only code, global data, heap, stack, etc)
- PC, $sp
- Opened file handles

A process needs certain resources, including CPU time, memory, files, and I/O devices.

The OS is responsible for the following activities for process management:
- Process creation and deletion
- Process suspension and resumption
- Provision of mechanisms for:
  - process synchronization
  - process communication

Parent process creates children processes, which, in turn creates other processes, forming a tree of processes.

Resource sharing:
- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Execution:
- Parent and children execute concurrently
- Parent waits until children terminate

Address space:
- Child duplicate of parent
- Child has a program loaded into it

UNIX examples:
- fork system call creates new process
- exec system call used after a fork to replace the process' memory space with a new program
C Program Forking Separate Process

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

Process Termination

- Process executes last statement and asks the operating system to decide it (exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination

Process State

- As a process executes, it changes state
  - new: The process is being created
  - ready: The process is waiting to be assigned to a process
  - running: Instructions are being executed
  - waiting: The process is waiting for some event (e.g. I/O) to occur
  - terminated: The process has finished execution

Diagram of Process State
Process Control Block (PCB)

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers (for context switch)
  - CPU scheduling information (e.g. priority)
  - Memory-management information (e.g. page table, segment table)
  - Accounting information (PID, user time, constraint)
  - I/O status information (list of I/O devices allocated, list of open files etc.)

---

The Linux process descriptor

Every process under Linux is dynamically allocated a task_struct, which contains all relevant information about a process.

- General
  - Process State, PID, Program name
  - Parent, youngest child, next sibling, previous sibling
  - Process Times (start_time, utime, stime, cutime, cstime)
  - Scheduling algorithm, priority, nice value, errno
- Owner: UID, EUID, SUID, FSUID, GID, EGID, SGID, FSGID
- Files
  - files: Information on all files opened by the process.
  - fs_struct: corresponding file system information
- Memory (mm_struct): Information about the memory (vma_struct, pgd) used by a process.
- IPC / Synchronization
  - Pointers to acquired Semaphores
  - Bitmask of received Signals and associated Signal handlers

---

```c
struct task_struct {
    [0] volatile long int state;
    [4] struct thread_info *thread_info;
    [8] atomic_t usage;
    [12] long unsigned int flags;
    [16] long unsigned int ptrace;
    [20] int lock_depth;
    [24] int load_weight;
    [28] int prio;
    [32] int static_prio;
    [36] int normal_prio;
    [40] struct list_head run_list;
    [48] struct prio_array *array;
    [52] short unsigned int ioprio;
    [56] unsigned int btrace_seq;
    [60] long unsigned int sleep_avg;
    [64] long long unsigned int timestamp;
    [72] long long unsigned int last_ran;
    [80] long unsigned int sched_time;
    [88] enum sleep_type sleep_type;
    [92] long unsigned int policy;
    ... other fields...

    // Some fields not shown for brevity
}
```
task_struct

struct task_struct {
    cpumask_t cpus_allowed;
    unsigned int time_slice;
    unsigned int first_time_slice;
    struct sched_info sched_info;
    struct list_head tasks;
    struct list_head ptrace_children;
    struct list_head ptrace_list;
    struct mm_struct *mm;
    struct mm_struct *active_mm;
    struct linux_binfmt *binfmt;
    long int exit_state;
    int exit_code;
    int exit_signal;
    int pdeath_signal;
    long unsigned int personality;
    unsigned int did_exec : 1;
    pid_t pid;
    pid_t tgid;
    struct task_struct *real_parent;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    struct task_struct *group_leader;
    struct pid_link pids[3];
    struct list_head thread_group;
    struct completion *vfork_done;
    int *set_child_tid;
    int *clear_child_tid;
    long unsigned int rt_priority;
    cputime_t utime;
    cputime_t stime;
    long unsigned int nvcsw;
    long unsigned int nivcsw;
    struct timespec start_time;
    long unsigned int min_flt;
    long unsigned int maj_flt;
    cputime_t it_prof_expires;
    cputime_t it_virt_expires;
    long long unsigned int it_sched_expires;
    struct list_head cpu_timers[3];
    uid_t uid;
    uid_t euid;
    uid_t suid;
    uid_t fsuid;
    gid_t gid;
    gid_t egid;
    gid_t sgid;
    gid_t ssgid;
    struct group_info *group_info;
    kernel_cap_t cap_effective;
    kernel_cap_t cap_inheritable;
    kernel_cap_t cap_permitted;
    unsigned int keep_capabilities : 1;
    struct user_struct *user;
    struct key *request_key_auth;
    struct key *thread_keyring;
    unsigned char jit_keyring;
    int oomkilladj;
    char comm[16];
    struct list_head cpu_times[3];
    uid_t tgid;
    struct task_struct *real_parent;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    struct task_struct *group_leader;
    struct pid_link pids[3];
    struct list_head thread_group;
    char comm[16];
    struct task_struct *real_parent;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    struct task_struct *group_leader;
    struct pid_link pids[3];
    struct list_head thread_group;
struct task_struct {
    void *security;
    struct audit_context *audit_context;
    seccomp_t seccomp;
    u32 parent_exec_id;
    u32 self_exec_id;
    spinlock_t alloc_lock;
    spinlock_t pi_lock;
    struct plist_head pi_waiters;
    struct rt_mutex_waiter *pi_blocked_on;
    void *journal_info;
    struct reclaim_state *reclaim_state;
    struct backing_dev_info *backing_dev_info;
    struct io_context *io_context;
    long unsigned int ptrace_message;
    siginfo_t *last_siginfo;
    wait_queue_t *io_wait;
    u64 rchar;
    u64 wchar;
    u64 syscr;
    u64 syscw;
    u64 acct_rss_mem1;
    u64 acct_vm_mem1;
    clock_t acct_stimexpd;
    struct cpuset *cpuset;
    nodemask_t mems_allowed;
    int cpuset_mems_generation;
    int cpuset_mem_spread_rotor;
    struct robust_list_head *robust_list;
    struct list_head pi_state_list;
    struct futex_pi_state *pi_state_cache;
    atomic_t fs_excl;
    struct rcu_head rcu;
    struct pipe_inode_info *splice_pipe;
    struct task_delay_info *delays;
};

struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    long unsigned int flags;
    long unsigned int status;
    __u32 cpu;
    int preempt_count;
    mm_segment_t addr_limit;
    void *sysenter_return;
    struct restart_block restart_block;
    long unsigned int previous_esp;
    __u8 supervisor_stack[0];
};

SIZE: 56

SIZE: 1408

Outline

1 Process Management
2 Context Switch
3 Summary
Context switches are one of the key techniques to allow multiple processes to share a single CPU. Basically, it is a procedure of storing and restoring the context state (including CPU registers, CR3, etc.) of a process (or a kernel thread) such that execution can be resumed from the same point at a later time.

A context switch could happen in a variety of cases in Linux/UNIX including:
- (I) Arbitrary places, when an asynchronous interrupt happens (could be timer) and the process has used its CPU time slice (preempted)
- (II) When a process voluntarily relinquishes their time in the CPU (e.g., invoking `sleep`, `waitpid` or `exit` system call)
- (III) When a system call is about to return
- (IV) Other system call subroutine places (besides system call return point), in which the kernel pro-actively checks whether a context switch is needed
- (V) In exception (e.g., page fault) handler
- (VI) When a system call gets blocked
### Process Management: Context Switch

#### Summary

When it happens a context switch (System Call)

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb8f000</td>
<td>sys_open</td>
<td>/fs/open.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>do_sys_open</td>
<td>/fs/open.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>do_filp_open</td>
<td>/fs/namei.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>do_path_lookup</td>
<td>/fs/namei.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>path_walk</td>
<td>/fs/namei.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>__link_path_walk</td>
<td>/fs/namei.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>do_follow_link</td>
<td>/fs/namei.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>switch_mm</td>
<td>/arch/x86/include/asm/mmu_context.h</td>
</tr>
</tbody>
</table>

When it happens a context switch (Interrupt Service)

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb8f000</td>
<td>do_page_fault</td>
<td>/arch/x86/mm/fault.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>handle_mm_fault</td>
<td>/mm/memory.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>__do_fault</td>
<td>/mm/memory.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>filemap_fault</td>
<td>/mm/filemap.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>lock_page</td>
<td>/include/linux/pagemap.h</td>
</tr>
</tbody>
</table>

### Process Management: Context Switch

#### Summary

When it happens a context switch (System Call)

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb8f000</td>
<td>sys_mmap2</td>
<td>/arch/x86/kernel/sys_i386_32.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>do_mmap_pgoff</td>
<td>/mm/mmap.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>mmap_region</td>
<td>/mm/mmap.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>do_munmap</td>
<td>/mm/mmap.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>remove_vma</td>
<td>/mm/mmap.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>schedule</td>
<td>/kernel/sched.c</td>
</tr>
</tbody>
</table>

When it happens a context switch (Interrupt Service)

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb8f000</td>
<td>do_fstat64</td>
<td>/fs/stat.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>cp_new_stat64</td>
<td>/fs/stat.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>copy_to_user</td>
<td>/arch/x86/lib/usercopy_32.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>switch_mm</td>
<td>/arch/x86/include/asm/mmu_context.h</td>
</tr>
</tbody>
</table>

When it happens a context switch (Interrupt Service)

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb8f000</td>
<td>page_fault</td>
<td>/arch/x86/mm/fault.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>handle_mm_fault</td>
<td>/mm/memory.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>__do_fault</td>
<td>/mm/memory.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>filemap_fault</td>
<td>/mm/filemap.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>lock_page</td>
<td>/include/linux/pagemap.h</td>
</tr>
<tr>
<td>fb8f000</td>
<td>schedule</td>
<td>/kernel/sched.c</td>
</tr>
<tr>
<td>fb8f000</td>
<td>push</td>
<td>/kernel/sched.c</td>
</tr>
</tbody>
</table>

When it happens a context switch (Interrupt Service)

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb8f000</td>
<td>c10bd884 syst_fstat64 381 /fs/stat.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c10bd244 cp_new_stat64 322 /fs/stat.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c119a3bc copy_to_user 853 /arch/x86/lib/usercopy_32.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c147bedc _cond_resched 6689 /kernel/sched.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c147b621 schedule 5413 /kernel/sched.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c147e875 do_page_fault 947 /arch/x86/mm/fault.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c10a29c5 handle_mm_fault 2991 /mm/memory.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c10a1136 __do_fault 2702 /mm/memory.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c108f4c3 filemap_fault 1499 /mm/filemap.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c108efab lock_page 314 /include/linux/pagemap.h</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c147bedc _cond_resched 6689 /kernel/sched.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c147b621 schedule 5413 /kernel/sched.c</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c1025ce4 switch_mm 35 /arch/x86/include/asm/mmu_context.h</td>
<td></td>
</tr>
<tr>
<td>fb8f000</td>
<td>c147e875 do_page_fault 947 /arch/x86/mm/fault.c</td>
<td></td>
</tr>
</tbody>
</table>
Process Management Context Switch Summary

When it happens a context switch (Other places)

System calls of getpid

Case-I: Arbitrary Places
Case-II: Voluntarily Relinquishing
Case-III: Syscall Return
Case-IV: Other Places in Syscall
Case-V: Exception Handler
Case-VI: Syscall Blocked

Interrupt Handler

sysenter/int 0x80

Exception Handler

Syscall Service Routine

Context Switch

sysexit/iretd

Context switch Profile of getpid

0 2 04 06 08 0 1 0 0 1 2 0

Case-V: Syscall Blocked
Case-V: Exception Handler
Case-IV: Other Places in Syscall
Case-III: Syscall Return
Case-II: Voluntarily Relinquishing
Case-I: Arbitrary Places