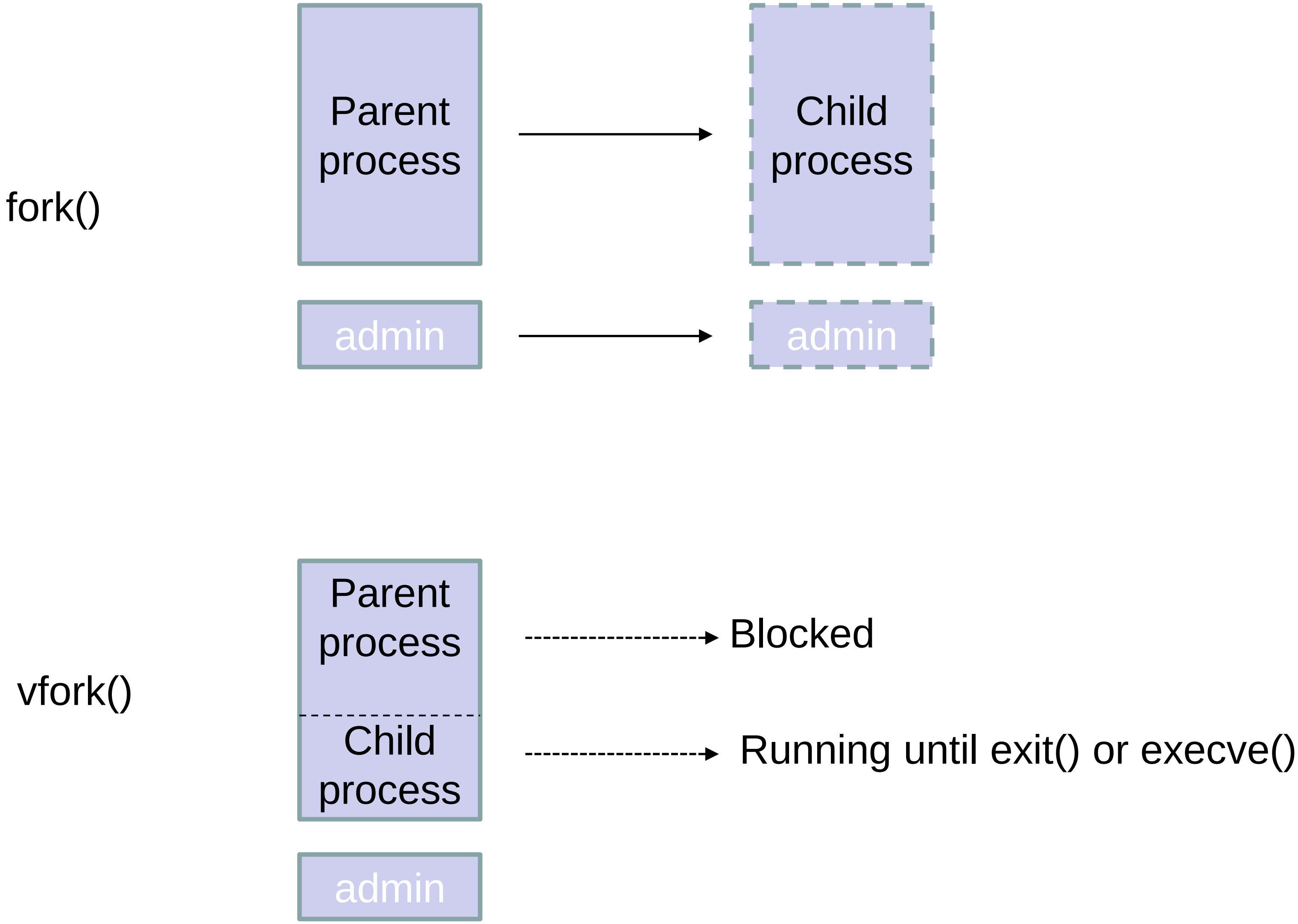
Lecture 8

Process management in Linux

A. Systems calls

- Linux addresses processes and threads as *tasks*.
- There are nearly 300 system calls, many related to process management. A (parent) process can create a new (child) process:
  - *fork(),* creates a child process as an exact copy of the parent process;
  - *vfork()* is a variation that eliminates the copy of the parent memory space in the case where fork() is quickly followed by an exec() call – new code is replacing the existing one. The child uses the parent memory space until invoking exec(). The parent is blocked during this time.
  - *clone()* allows to specify which of the parent's resources are to be shared with the child and which are to be copied;
  - *execve()* allows a process to specify a program to begin running in place of the current one.


fork()
Parent process
Child process
admin
admin
vfork()
Parent process
Child process
Blocked
Running until exit() or execve()
admin

- *exit()* is for process termination. Applications that are ready to finish generally either return from the main() function or call exit(). Exit() performs some application-level cleanup of open files and then issues the _exit() call. The kernel then frees the process' resources and makes an exit status available to the parent process.
- *kill()* is the means by which a process sends a signal – for some signals, the default behaviour is to terminate the process, for most there is a signal handler that is invoked when the signal is received.
- *wait4()* and *waitpid()* allow a parent process to inquire as to the state of a child. Their purpose is to notify the parent when the child has exited and to deliver its exit status to the parent.
- *nice()* gives the process the ability to adjust its priority level – higher values represent lower priorities.

Calls to the scheduler

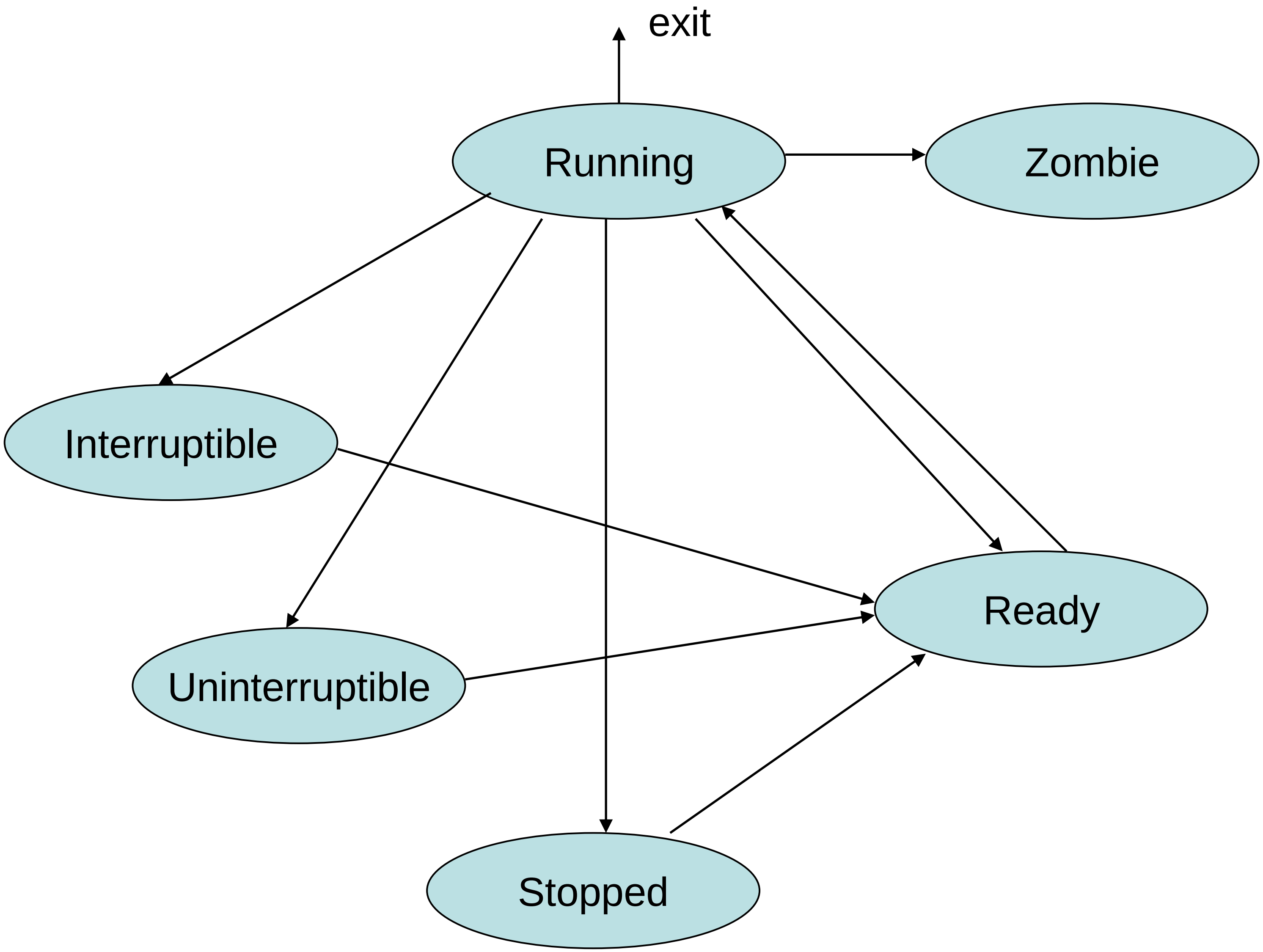
- *sched_setscheduler()* allows a process with enough privileges to change the policy and priority level the scheduler uses for the specified process.

- *sched_getscheduler()* allows the process to query which scheduling policy is currently in use.

- *sched_yield()* allows a process to give up the remainder of its current time slice.

B. Process states

- TASK_RUNNING refers to both running and ready;
- TASK_INTERRUPTIBLE represents a blocked state of a process that can be awaken by signals sent by other processes;
- TASK_UNINTERRUPTIBLE represents a blocked state from where processes do not come out in response to signals.
- TASK_STOPPED is a state of a process that received the signal SIGSTOP, from which it comes out when it received SIGCONT. These signals implement task control in several of the user interface shells.
- TASK_TRACED is used as part of the tracing facility where a process can control the execution of another process. It is used for the implementation of debuggers.
- EXIT_ZOMBIE
- EXIT_DEAD is used when a process terminates and its parent is not notified. The process can be removed from the system immediately.


exit
Running
Zombie
Interruptible
Ready
Uninterruptible
Stopped

B. Process creation

- In Linux, processes can be created by either *fork()*, or *vfork()*, or *clone()*.
- All three call *do_fork()*. This function has three responsibilities:
  1. *do_fork()* calls *copy_process()* for creating the child; it uses the *clone_flags* parameter to determine which of parent's resources are copied and which are shared.
  2. sets up the suspension of the parent process if it's about vfork().
  3. sets up the initial state of the child; it can be either in a Ready or Stopped state.

- Handling the system call
  - The clone_flags long integer parameter is treated as a set of one-bit flags that control the copy vs share.

```
long do_fork(unsigned long clone_flags, unsigned long stack_start, struct pt_regs
    *regs, unsigned long stack_size,…)
{
    struct task_struct *p;
    int trace = 0;
```

  - In most cases, stack_start is the calling process SP. The regs parameter points to a structure containing the machine registers saved on entry to the system call.

Allocating the process ID

- alloc_pid() maintains a global variable called last_pid, which is the most recently assigned pid.
- When called, it tries last_pid+1. Because pid can be reused, it must be checked if the tentative pid is in use. To make the check quickly, alloc_pid() maintains a bitmap of all possible pids. If the pid is in use, the search will start from there for the next unused one.
- If the pid allocation fails, -EAGAIN is returned to indicate the failure; however allocation can be tried later, as process termination returns valid pid.
- If one unused value was found, it will be returned and assigned to pid.

```
struct pid *pid = alloc_pid();
int nr;
if ( ¬pid)
    return –EAGAIN;
nr = pid →nr;
if (unlikely(current→ptrace)) {
    trace = fork_traceflag(clone_flags);
    if (trace)
        clone_flags |= CLONE_PTRACE;
}
```

Unlikely() as likely() are macros that indicate to the compiler the expectation of an if condition to evaluate to true or to false.

Creating the child process

- Now, *copy_process()* can be called to do most of the work for creating the child process.
- The key passed arguments are clone_flags, stack_start, regs and pid.
- At this point, only the parent returns.
- The child is set up to go directly to the code that returns from the process creation call.

p = copy_process(clone_flags, stack_start, regs, stack_size,..,nr);

Starting the child process

- The child inherits the parent's Ready state. By calling wake_up_new_task(), the child is inserted into the appropriate ready queue.
- However, it is possible to create a process that starts in the Stopped state; state that can be changed to TASK_STOPPED.

```
if (¬(clone_flags & CLONE_STOPPED))
    wake_up_new_task(p, clone_flags);
else
    p →state = TASK_STOPPED;
if (unlikely(trace)) {
    current→ptrace_message = nr;
    ptrace_notify((trace <<8)|SIGTRAP);
}
```

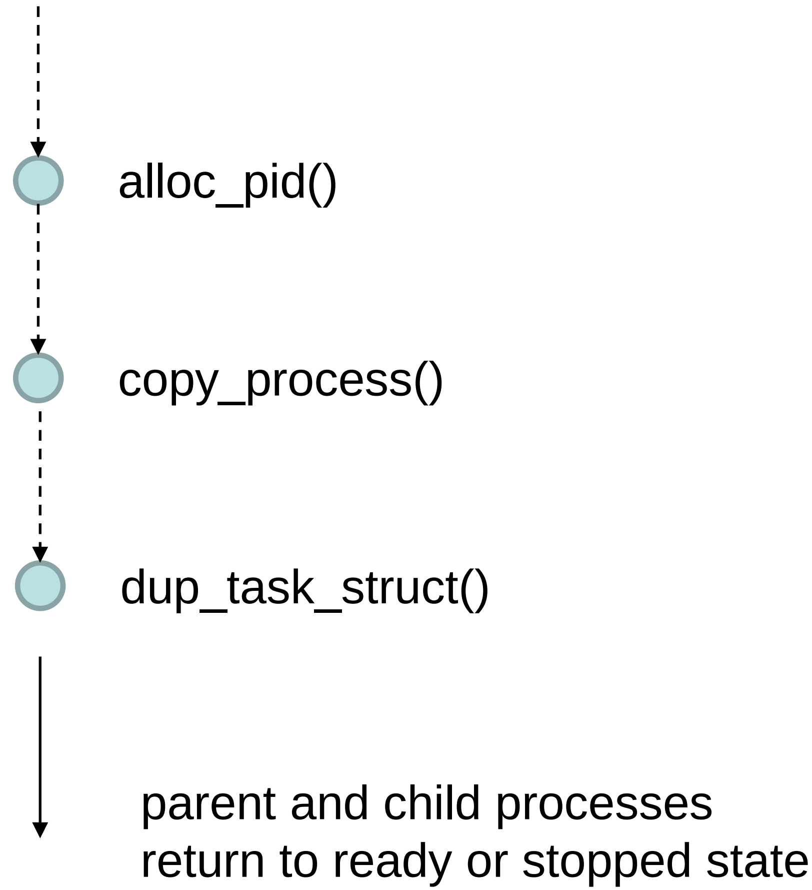
Determining parent behaviour

- In the case of *vfork()*, the parent is blocked until the child issues an _exit() or an execve() call.
- The call *wait_for_completion()* sets the state of the parent to TASK_UNINTERRUPTIBLE in order to block it. When the vfork structure is modified, indicating that the child finished using the parent's memory space, the parent is moved from Blocked to Ready.

```
if (clone_flags & CLONE_VFORK) {
    wait_for_completion(&vfork);
    if (unlikely(current→ptrace&PT_TRACE+VFORK_DONE))
            ptrace_notify((PTRACE_EVENT_VFORK_DONE<<8)|
    SIGTRAP);
    }
```

Creating the process admin part

- copy_process(), along with the functions it calls, does the real work of creating the new process.
- The call to *dup_task_struct()* allocates a new task structure and copies the parent's structure into it. It also sets up the pointers between the process stack and the new task structure. At this point, we have a new process table entry for the child process with an initial set of values. Many of the members of this structure are changed later in this function.
- After initializing some values, locks and timers, there are lines of code that handle the copying or sharing of the parent's resources – memory, files.
- The last one, copy_thread() handles the difference in the way the child returns.
- sched_fork() splits the remainder of the parent's time slice evenly between the parent and the child. We don't want other processes starved by one that continuously creates children.
- The call to fork_out(): if errors were encountered during the process, the error will be returned. Otherwise, the pointer to the process table entry will be returned.


alloc_pid()
copy_process()
dup_task_struct()
parent and child processes
return to ready or stopped state

Questions

1. What are the main differences between fork(), vfork() and clone()? What criteria are used to choose one call or another?
2. With vfork(), the parent process is suspended until the child process either exits or calls exec(). How is the parent process returning to the ready state?
3. How can the operation of finding an available PID be accelerated?
4. Is the child process admin part different from that of its parent process?