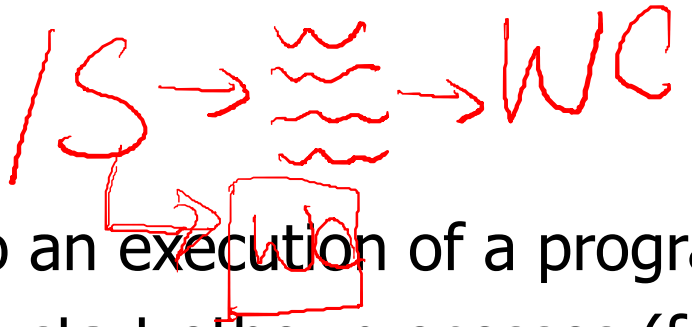
Process – Part 1

Process (1)

- In UNIX system,
  - Shell creates a new process each time it starts up a program
    - $ cat file1 file2
    - $ ls | wc -l
  - Processes correspond to an execution of a program
  - Any process may in turn start other processes (fork/exec model)
  - System calls for process creation and manipulation
    - fork()
      - Create a new process by duplicating the calling process



Process (2)

- exec()
  - Perform the transformation of a process by overlaying its memory space with a new program
  - One system call and a family of library
- wait()
  - Provide rudimentary process synchronization
  - Allow one process to wait until another related process finishes
- exit()
  - Terminate a process

Process ID

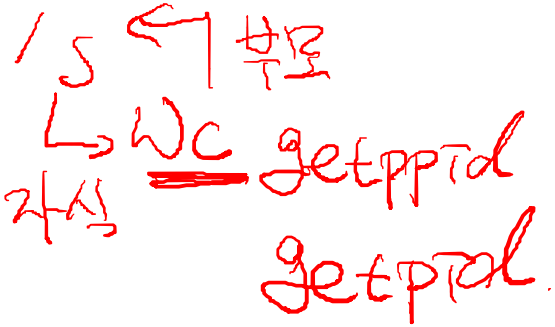
- Process ID
  - Every process has a unique process ID.
  - The index of the process table entry in the kernel.
  - Often used as a piece of their identifiers, to guarantee uniqueness.

Getting Process ID

- #include <sys/types.h>

  #include <unistd.h>

  pid_t getpid(void);
  - Returns the process ID of the current process.
- pid_t getppid(void);
  - Returns the process ID of the parent of the current process.



Linux Swapper Process

- pid = 0
- The ancestor of all processes.
- A kernel thread created from scratch during the initialization phase of Linux.
- Initializes all the data structures needed by the kernel, enables interrupts and created another kernel thread, named *init process*.
- After creating the init process, swapper repeatedly executes *cpu_idle()* function.
- Swapper is selected by the scheduler when there is no other processes in the running state.

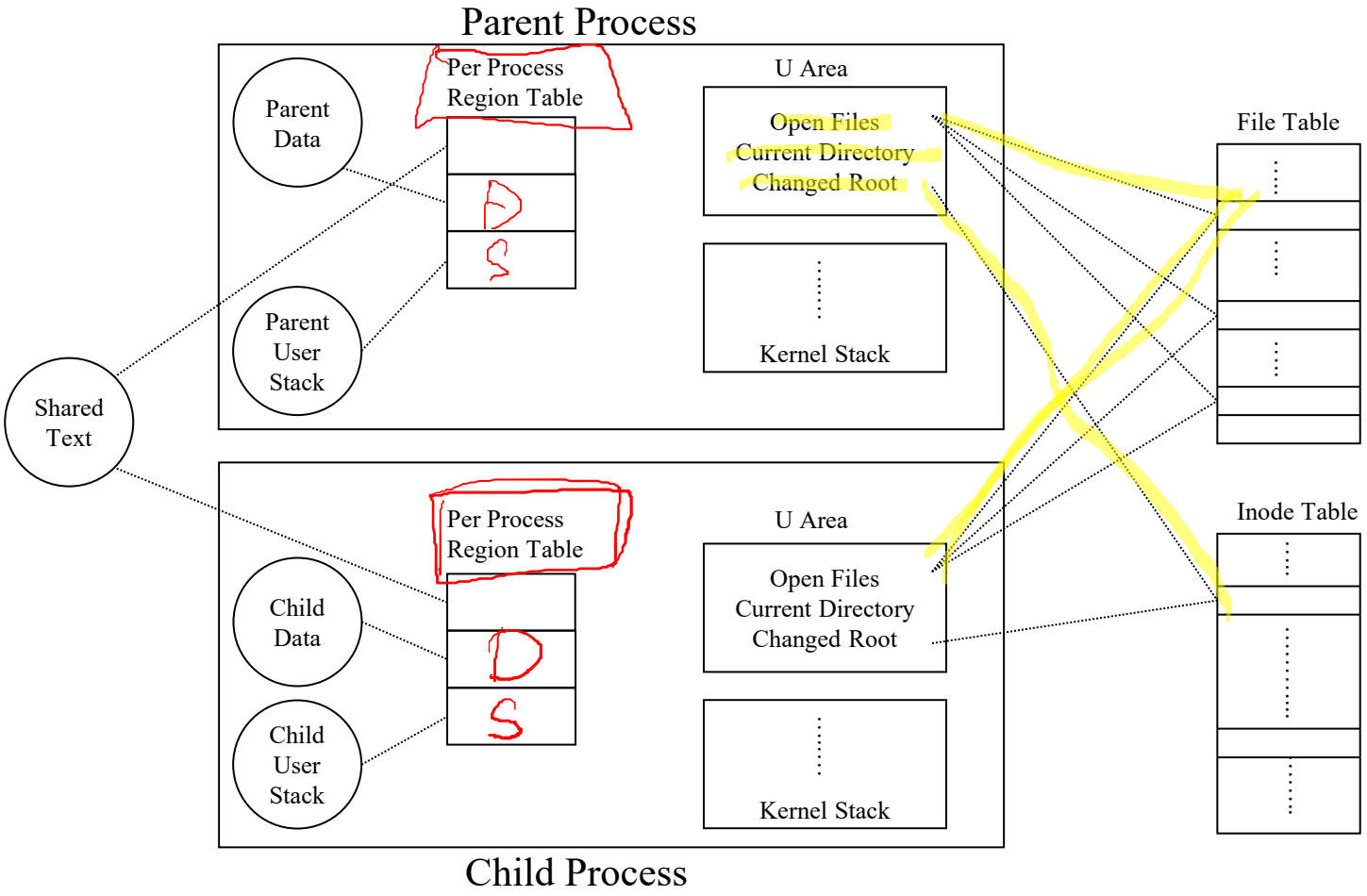
Linux Init Process

- pid = 1
- *init()* invokes the execve( ) system call to load the executable program /sbin/init. As a result, the init kernel thread becomes a regular process having its own per-process kernel data structure (*init* is a normal user process with super-user privileges).
- It never terminates, since it creates and monitors the activity of all user processes.

fork (1)

- #include <sys/types.h>
  #include <unistd.h>
  pid_t fork(void);
  - The only way a new process is created is when an existing process calls the fork( ) system call.
  - The new process created by fork is called the child process.
  - fork( ) is called once but returns twice.
    - Parent: pid of child process
    - Child: 0
  - Both the child and parent continue executing with the instruction that follows the call to fork( ).
  - The child is a copy of parent.
    - Child gets a copy of the parent's data space, heap, and stack.
    - Often the parent and child share the text segment.

fork (2)


Parent Process
Per Process
Region Table
U Area
Open Files
Current Directory
Changed Root
Parent
Data
Parent
User
Stack
Kernel Stack
Shared
Text
File Table
Inode Table
Per Process
Region Table
U Area
Open Files
Current Directory
Changed Root
Child
Data
Child
User
Stack
Kernel Stack
Child Process

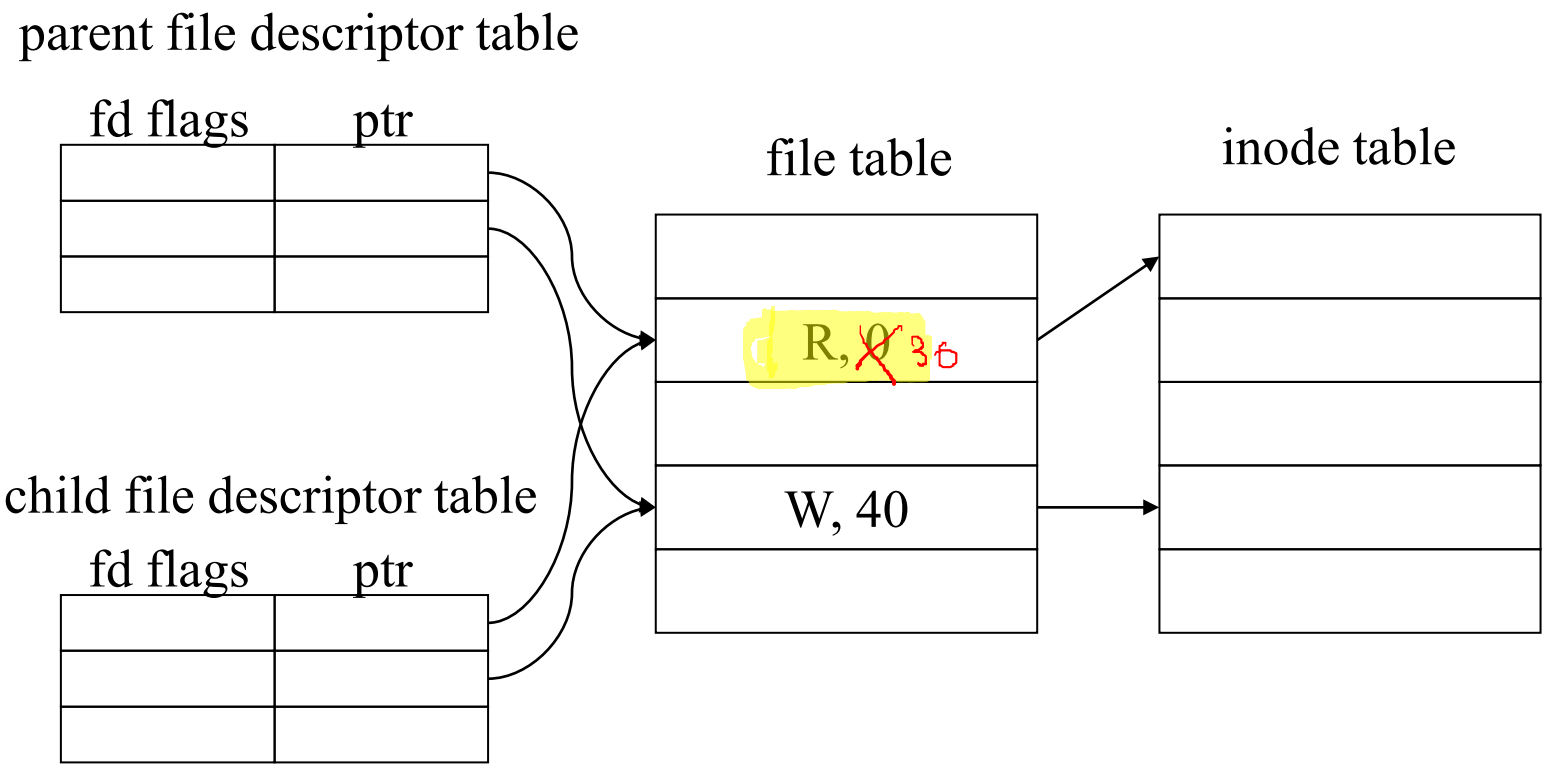
fork (3)

- The child process *inherit* the following from the parent.
  - Open files, file mode creating mask.
  - Real-uid (gid), effective-uid (gid), set-user (group)-id flag
  - Process group-id, session-id, terminal control
  - Current working directory, current root directory, environment
- Difference between the parent and child.
  - PID and PPID
- We never know if the child starts executing before the parent or vice versa.
  - Depends on the scheduling algorithm used by the kernel.
  - The synchronization may be required.
- File sharing
  - All descriptors that are open in the parent are duplicated in the child.

fork (4)

- Two cases for handling the descriptors after a fork.
  - The parent waits for the child to complete, if the parent does not need to do anything with its descriptors.
    - When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
  - The parent and child each go their own way.
    - After the fork, the parent closes the descriptors that it doesn't need and the child does the same thing.
    - This way neither interferes with the other's open descriptors.

fork (5)


parent file descriptor table
fd flags
ptr
file table
inode table
R, 0
36
child file descriptor table
fd flags
ptr
W, 40

fork (6)

- Two main reasons for fork to fail.
  - If there are already too many processes in the system.
  - If the total number of processes for this real user ID exceeds the system's limit.
- Two uses for fork
  - A process wants to duplicate itself.
    - Parent and child can each execute different sections of code at the same time.
  - A process wants to execute a different program.
    - Common for shells.
    - The child does an exec( ) right after it returns from the fork( ).

Example #1: fork

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void main(void)
{
   pid_t pid;

   printf("Hello, my pid is %d\n",
          getpid());

   if ((pid = fork()) == 0) {
      /* child process */
      printf("child: pid = %d, ppid = %d\n",
             getpid(), getppid());
   }
   else {
      /* parent process */
      printf("parent: I created child with
             pid=%d\n", pid);
   }

   /* Following line is executed by
      both parent and child */

   printf("Bye, my pid is %d\n", getpid());
}
```

113

114

115

	fork()	getpid	getppid
114 Parent	115	114	113
115 Child	0	115	114

clone(), fork() and vfork()

- Overheads of fork() in traditional UNIX system
  - Resources owned by the parent process are duplicated in the child process
    - Slow and inefficient
- Three different mechanism to overcome this problem
  - Copy On Write
    - Allow both the parent and the child to read the same physical pages
    - Whenever either one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process
  - Lightweight process
    - Allow both the parent and the child to share many per-process kernel data structures, such as the paging table, open file table and others
  - vfork()
    - Create a process that shares the memory address space of its parent
    - Parent's execution is blocked until the child exits or executes a new program to prevent the parent form overwriting data needed by the child

clone (1)

- Linux clone() system call
  - Create a lightweight process.
  - Generalized form of fork() and pthread_create() that allows the caller to specify which resources are shared between the calling process and the newly created process
  - Require you to specify the memory region for the execution stack that the new process will use
  - Should not ordinarily be used in programs
  - Use fork() to create new processes or pthread_create() to create threads
  - fork() and pthread_create() is implemented by Linux as a clone() system call

clone (2)

- #include <sys/types.h>

  #include <unistd.h>

  pid_t clone(void *_child_stack_, unsigned long _flags_);
  - _child_stack_
    - Specifies the user mode stack pointer.
    - If it is equal to 0, the kernel assigns to the child the current parent stack pointer, i.e., temporarily the share user mode stack. Because of the Copy On Write mechanism, they usually get separate copies of the user mode stack as soon as one tries to change the stack.

clone (2)

- *flags*
  - The low 1 bytes specifies the *signal number* to be sent to the parent process when the child terminates; the SIGCHLD signal is generally selected.
  - The remaining 3 bytes specify the resources *shared* between the parent and the child process.
    - CLONE_VM (memory descriptor, page tables), CLONE_FS (root and current directory table), CLONE_FILES (open file table), CLONE_SIGHAND (signal handler), CLONE_PID (share PID), CLONE_PTRACE (process tracing), CLONE_VFORK (vfork( ) system call)

vfork

- #include <unistd.h>

  #include <sys/types.h>

  pid_t vfork(void);
  - Semantic of vfork( ) differs from fork( )
    - Intended to create a new process when the purpose of the new process is to exec( ) a new program.
    - vfork( ) creates the new process just like fork( ), without fully copying the address space of the parent into the child.
    - The child just calls exec( ) or exit( ) right after the vfork( ).
    - While the child is running, until it calls either exec( ) or exit( ), the child runs in the address space of the parent.
      - Provides an efficiency gain.
    - vfork( ) guarantees that the child runs first, until the child calls exec( ) or exit( ).

Example #2: vfork

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int glob = 6;

int main(void)
{
   int var;
   pid_t pid;

   var = 88;
   printf("before vfork\n");
   /* we don't flush stdio */

   if ((pid = vfork()) < 0) {
      perror("vfork error");
      exit(1);
   }
   else if (pid == 0) { /* child */
      glob++;
      /* modify parent's variables */
      var++;
      _exit(0);
      /* child terminates. why exit? */
   }
   /* parent */
   printf("pid = %d, glob = %d, var
      = %d\n",
            getpid(), glob, var);
   exit(0);
}
```

glob = 6

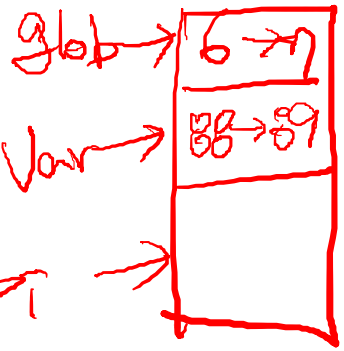
Parent

Var = 88

pid = 100

child = 101

Parent, child



100, 7, 89