Process Creation and Termination

Process Creation

Only way to create a new process in UNIX is to invoke the `fork()`system call. The process that invokes `fork()`is called the parent process, and the newly created process is called the child process. Syntax:

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

On return, the two processes have identical copies of their user-level context except for the return value `pid`.

In the parent process, `pid` is the child process ID.
In the child process, `pid` is 0.
The only process not created via `fork` is process 0 internally by the kernel.

The kernel does the following:

- **Allocates a spot in the process table** for the new process.
- **Assigns a unique ID number to child process** (not 0).
- **It makes a logical copy of the context of the parent process**. Since certain portions, like the text region, may be shared between the processes, it can sometimes increase a region reference count instead of copying the region to a new physical location in memory.
- **It increments the file and inode table counters** for the files associated with the processes.
- **It returns the ID number of the child process to the parent process**, and a 0 value to the child process.

Algorithm for fork

Assumption: Traditional swapping system and system has enough main memory to store the child process.

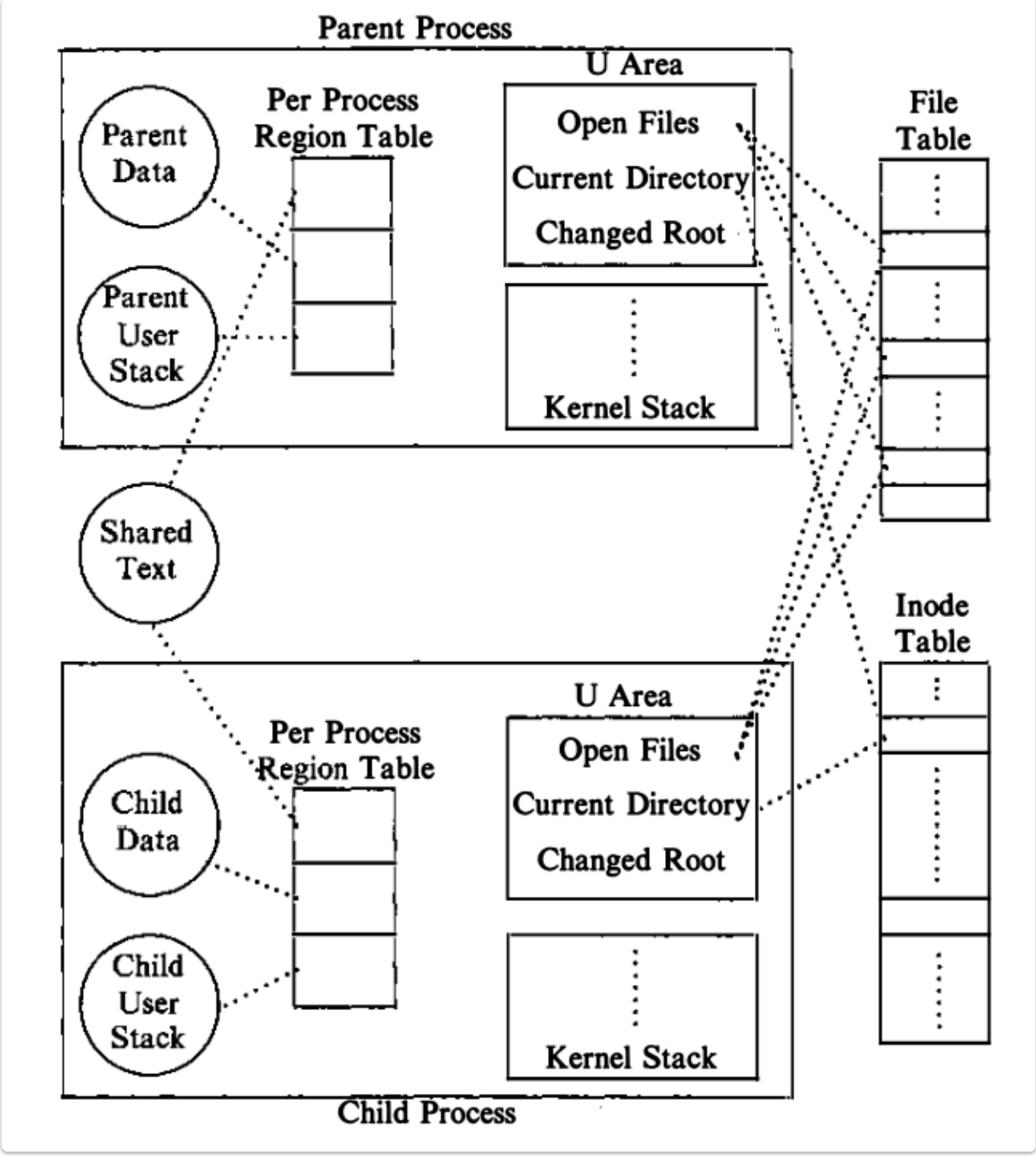
```
algorithm fork
input: none
output: to parent process, the child PID
                to child process, 0
{
```

```
        check for available kernel resources;
        get free proc table slot, unique PID;
        check that user not running too many process;
        mark child state "being created";
        copy data from parent proc table slot to new child slot;
        increment counts on current directory inode and changed root (if
applicable);
        increment open file counts in file table;
        make copy of parent context (u area, text, data, stack) in memory;
        push dummy system level context layer onto child system level context;
                                dummy context contains data allowing child process
to recognize itself,
                                and start running from here when scheduled;
        if (executing process is parent process){
                change child state to "ready to run";
                return (child ID); /* from system to user */
        }
        else{
                initialize u area timing fields;
                return 0; /* to user */
        }
}
```

Selecting a unique PID: If ID numbers reach a max value, it starts from 0 again. Since most process execute for a short time, most ID number are not in use when ID assignment wraps around.

The kernel initializes the child's process table slot, copying various fields from the parent slot. For instance, the child "inherits" the parent process **real and effective user ID numbers**, the **parent process group**, and the **parent nice value**, used for calculation of scheduling priority.

Not only does the child process inherit access rights to open files, but it also shares access to the files with
the parent process because both processes manipulate the same file table entries.



The processes have **identical copies** of the text, data, and (user) stack regions; the region type and the
system implementation determine whether the processes can share a physical copy of the text region.

Example

```
#include <fcntl.h>
int fdrd, fdwt;
char c;

main(int argc, char** argv){
        if(argc != 3)
                exit(1);
        if((fdrd = open(argv[1], O_RDONLY)) == -1)
                exit(1);
        if((fdwt = creat(argv[2], 0666)) == -1)
                exit(1);

        fork();
        /* both procs execute the same code */
        rdwrt();
        exit(0);
}

rdwrt(){
        for(;;){
                if(read(fdrd, &c, 1) != 1)
                        return;
                write(fdwt, &c, 1);
        }
}
```

The file descriptors in both processes refer to the same file table entries.

Consider the following scenario where the processes are about to read the two character sequence "ab" in the source file. Suppose the parent process reads the character 'a', and the kernel does a context switch to execute the child process before the parent does the write. If the child process reads the character 'b' and writes it to the target file before the parent is rescheduled, the target file will not contain the string "ab" in the proper place, but "ba".

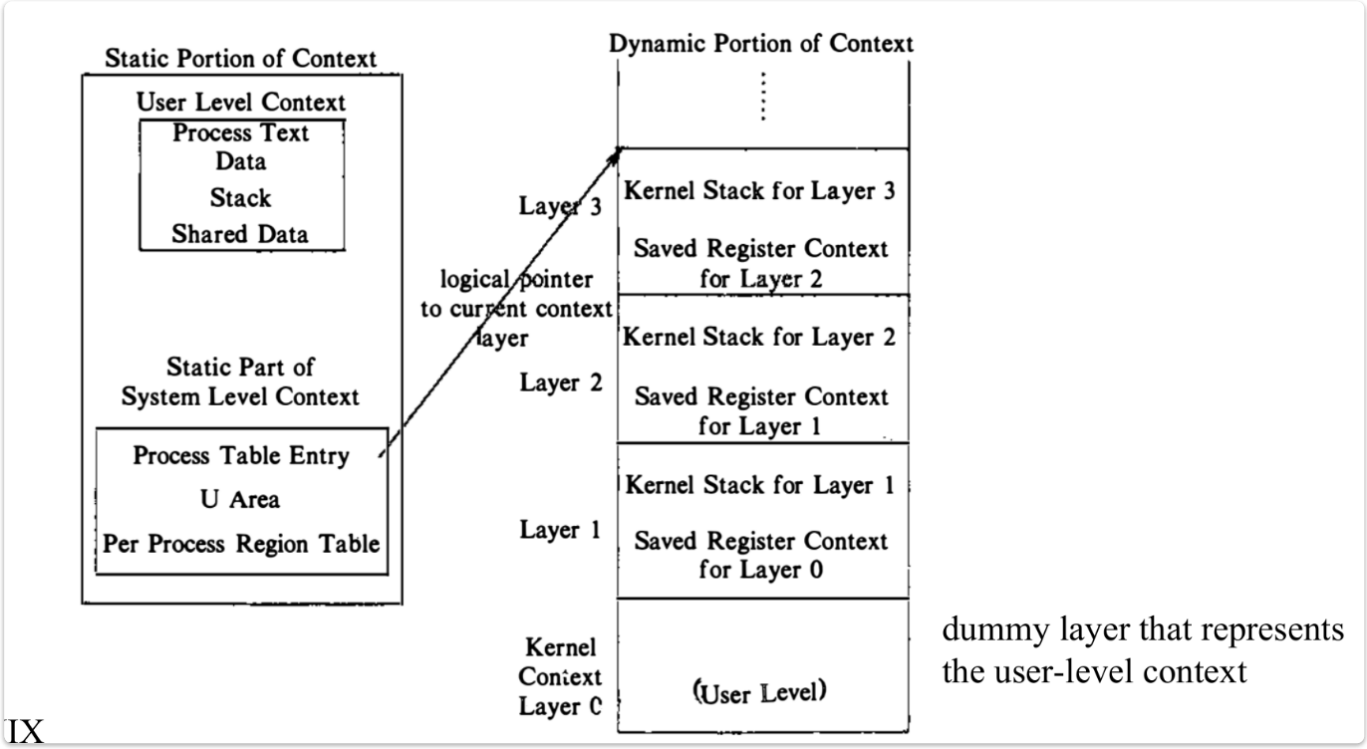
Process Context

Process context consists of:

- Contents of its (user) address space
- The content of its hardware registers
- Kernel data structures that relate to the process

or alternatively,

- User level context
  - The process **text, data, user stack and shared memory** that occupy the virtual address space of the process
  - **Parts of the virtual address space** of a process that **periodically do not reside in the main memory** because of swapping and paging.
- register context
  - The **program counter**
  - The **process status register** (PS)
  - The **stack pointer**
  - **General purpose registers** containing data generated by the process during its execution.
- system level context
  - **Static part**
    - The **process table entry of a process** defines the state of the process, and contains control information
    - The **u area** of a process contains control information that needs to be accessed only in the context of the process
    - **Pregion entries, region tables and page tables**, define the mapping from virtual to physical addresses and therefore define the text, data, stack and other regions of a process.
  - Dynamic part: viewed as a stack of context layers
    - The **kernel stack contains the stack frames of kernel procedures** as a process executes in kernel mode.
    - **A set of layers, visualized as a last-in-first-out stack.**



IX

Handling Interrupts

```
algorithm inthand /* handle interrupts */
input: none
output: none
{
        save (push) current context layer;
        determine interrupt source;
        find interrupt vector;
        call interrupt handler;
        restore (pop) previous context layer;
}
```

The kernel stack is used for storing interrupt handler stack frames.

INTERRUPT VECTOR

Interrupt Number	Interrupt Handler
0	clockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	otherintr

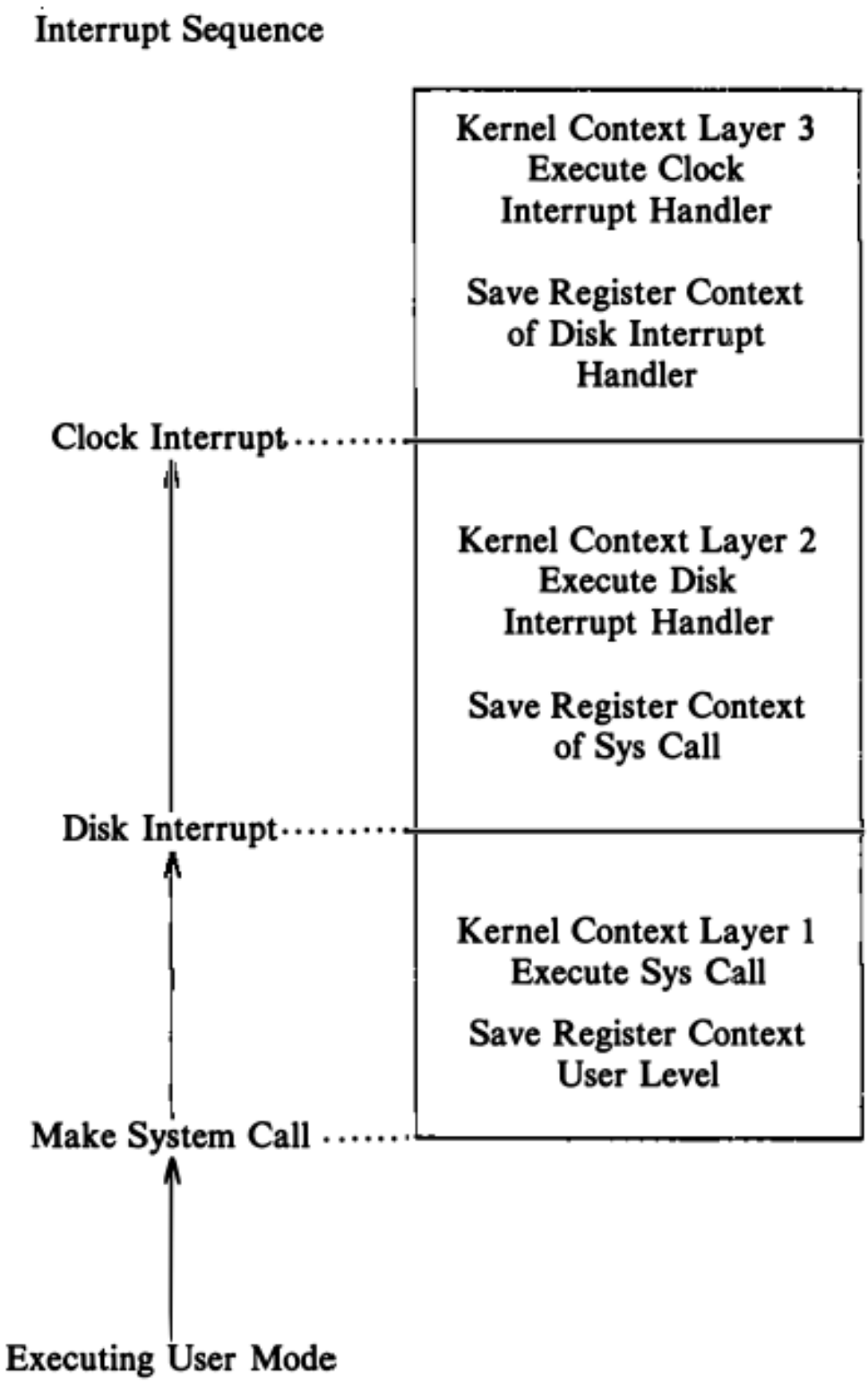


Figure 6.11. Example of Interrupts

kernel context layer is used for the current interrupt, and register context is of the previous layer that contains the return address, etc.

Process Termination

Processes terminate on UNIX system by executing the `exit()` system call. An *exiting* process enters the zombie state, relinquishes its resources, and dismantles its context except for its slot in the process table, hence why it is called a zombie process. Syntax:

```
exit(status);
```

where the value of `status` is returned to the parent process for its examination. Processes may call `exit()`implicitly at the end of the program or explictly. The startup routine linked with all C programs calls `exit()`when the program returns from the `main` function.

The system imposes no time limit on the execution of the program. For instance, process 0 (swapper) and process (`init`) exist throughout the lifetime of a system.

Algorithm for exit

```
algorithm exit
input: return code for parent process
output: none
{
        ignore all signals;
        if (process group leader with associated control terminal){
                send hangup signal to all members of process group;
                reset process group for all members to 0;
        }
        close all files (internal version of algorithm close);
        release current directory (algorithm iput);
        release current (changed) root, if exists (algorithm iput);
        free regions, memory associated with process (algorithm freereg);
        write accounting record;
        make process state zombie;
        assign parent process ID of all child processes to be init process (1);
                if any children were zombie, send death of child signal to init;
        send death signal to parent process;
        context switch;
}
```

Important Points:

- The kernel resets the process group number to 0 for processes in the process group, because it is possible that another process will later get the process ID of the process that just exited and that it too will be a process group leader.

- The kernel also writes an accounting record to a global accounting file, containing various run-time statistics such as user ID, CPU and memory usage, and amount of I/O for the process.
- the kernel disconnects the process from the process tree by making process 1 (init) adopt all its child processes.