



Unix Process Management

31/01/2004



Unix Process Management

- ◆ Operating system functions executes within user process.
- ◆ 2 modes of execution
 - User mode and Kernel mode
- ◆ 2 types of processes are available
 - System processes(Execute OS code)
 - User processes(Execute user program code).
- ◆ System call is used to transfer from user mode to system mode.

Process states in UNIX

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

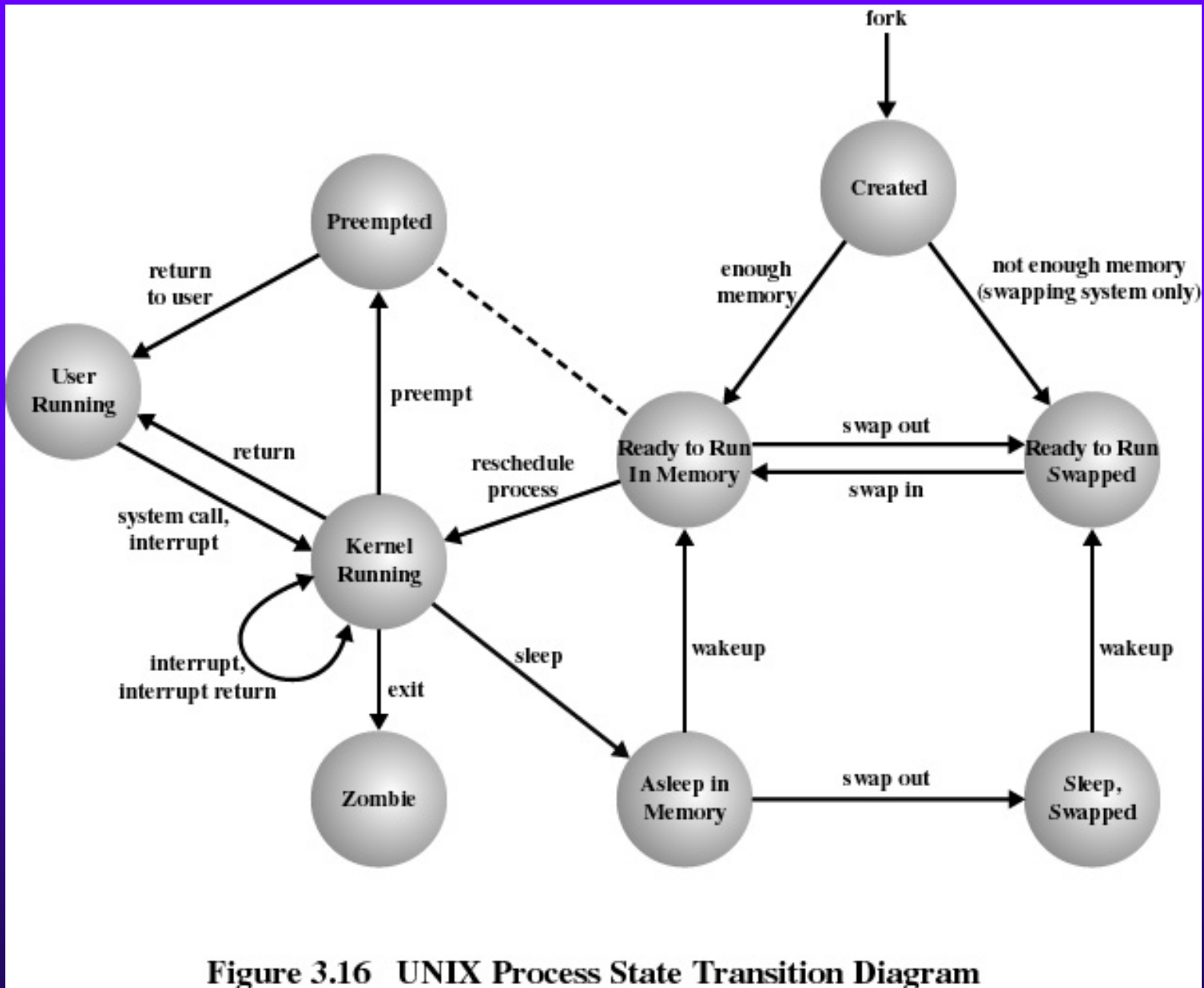



Figure 3.16 UNIX Process State Transition Diagram

- 
- ◆ Preemption can only occur when the process is about to move from Kernel mode to User mode.
 - ◆ While the process is running in Kernel mode it may not be preempted. This make UNIX not suitable for real time processing.
 - ◆ Process 0 is a special process that is created when the system boots. Process 1 (init process) is the child of Process 0. All the other processes in UNIX has Process 1 as ancestor. All new processes are created under Process 1 (init process).



Process Description

- ◆ Elements of process image -- divided into 3
 - User level context
 - Process text, Process data, User stack and Shared memory.
 - Register context
 - Program counter, Process status registers, stack pointer, general purpose registers...
 - System level context
 - Process table entry, User area, Per process region table, Kernel stack
 - When a process is not running the processor status information is stored in register context area.

Unix Process Table Entry


- ◆ Process status
- ◆ Pointers – to user area & process memory area (text, data, stack)
- ◆ Process size – enables OS to know how much space to allocate
- ◆ User identifier
 - Real user ID \rightarrow ID of user who is responsible for the process
 - Effective user ID \rightarrow used by process to gain temporary privilege, while the program is being executed as a part of process.
 - Process identifiers \rightarrow ID of the process



- ◆ Event descriptor → valid when a process is in sleep state. When the event occurs, the process is transferred to a ready to run state.
- ◆ Priority → used for scheduling
- ◆ Signal → enumerates signals send to process but not yet handled.
- ◆ Timers → Process execution time, kernel resource utilization, and user set timer used to send alarm signal to a process.
- ◆ P_link → pointer to the next link in the ready queue.
- ◆ Memory status → indicates the process image is in the main memory or swapped out.

UNIX User Area

- ◆ Process table pointer → Indicates entry corresponding to the user area.
- ◆ User identifiers → Real & Effective user Ids, used to determine user privileges.
- ◆ Timers → Record time that the process spent executing in user mode & kernel mode.
- ◆ Signal handling array → For each type of signal defined in the system, indicate how the process will react to receipt of that signal.
- ◆ Control terminal → Indicate login terminal for this process, if exist.
- ◆ Error field → Record errors encountered during system call.

- 
- ◆ Return value → Contain result of s/m call
 - ◆ I/O parameters → Describes amount of data transfer, the address of the source data array in user space, file offset for I/O.
 - ◆ File parameters → Current directory & current root describe the file system environment of the process.
 - ◆ User file descriptor table → Record the file the process has open
 - ◆ Limit fields → Restrict the size of the process & size of the file it can write.
 - ◆ Permission mode field → Mask mode settings on files the process creates.

System level context

◆ 2 parts

- 1. Static (Process table entry, User area, Per process region table).
- 2. Dynamic (Kernel stack)
- Process table entry contains process control information that is accessible to the kernel to the kernel at all time. So in VM systems all process table entries are maintained in main memory.
- User area contains additional process control information that is needed by the kernel when it executes in context of a process. It is also used when swapping process to and from the memory.
- Process reg. table is used by memory management s/m
- Kernel stack – used when the process is executing in kernel mode and contain information that must be saved and restored as procedure calls and interrupts




Fork program in 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);
    }
}
```

Fork() in UNIX

- ◆ Allocate slot in the process table for the new process
- ◆ Assigns a unique process id to the new process
- ◆ Make a copy of the process image of the parent, with the exception of shared memory
- ◆ It increases counters for any files owned by the parent, to reflect that an additional process now also owns these files.
- ◆ It assigns the child process to a ready to run state
- ◆ It returns the ID number of the child to the parent process and a 0 value to the child process.
- ◆ These all works are done in Kernel of parent process.



◆ After completing those functions OS will do the following operations as a part of dispatcher routine

- Stay in the parent process. Control returns to the user mode at the point of the fork call of the parent.
- Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
- Transfer control to another process. Both child and parent are left in the ready to run state.



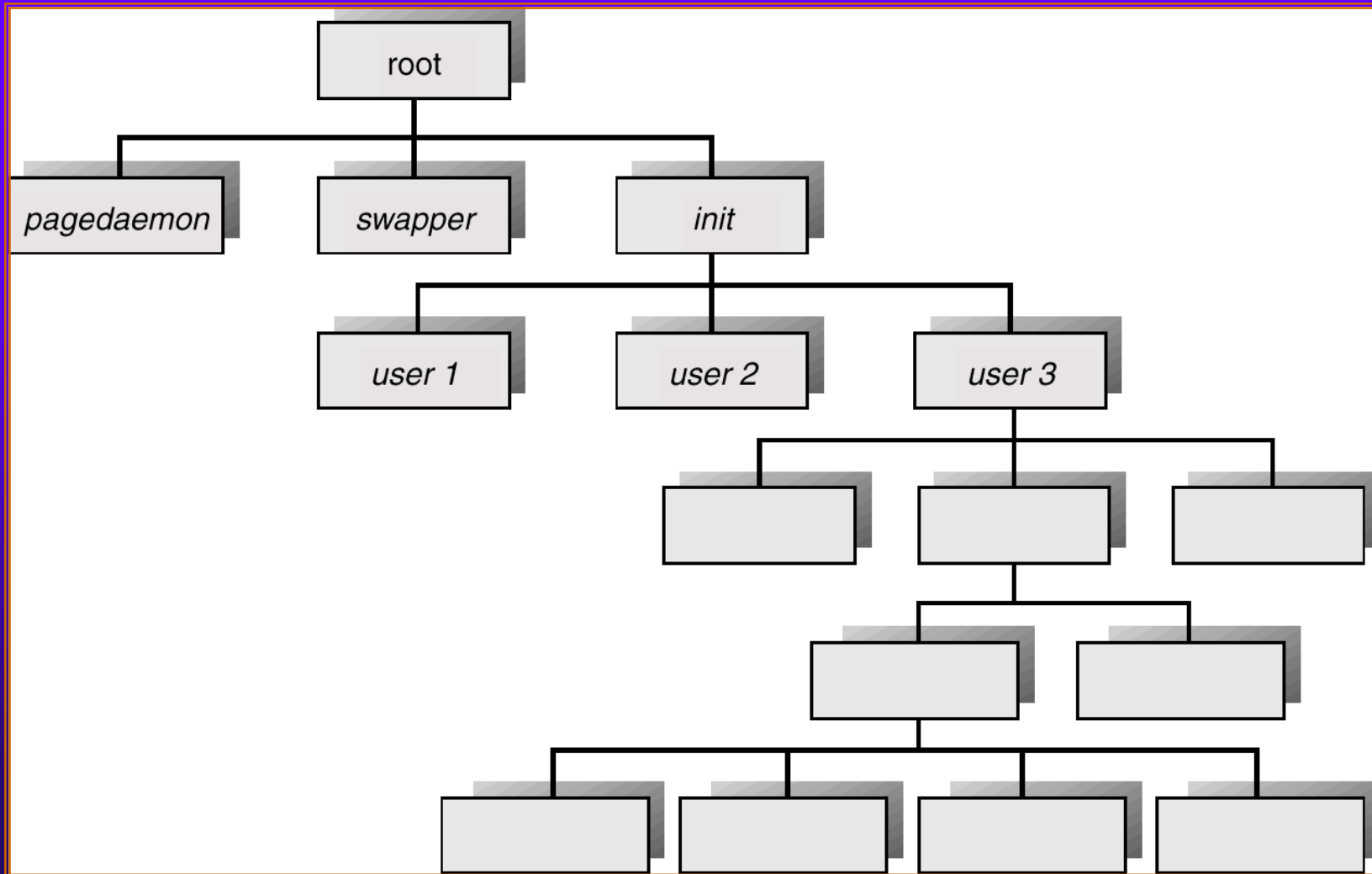
Types of processes in UNIX

- ◆ Mainly 3 types

- User process
- Daemon process
- Kernel process
- Daemon process

- Performs the function in a system wide basis. The function can be of any auxiliary kind, but they are vital in controlling the computational environment of the system.
- Example Print spooling, Network Management.
- Once created Daemon process can exist throughout the life time of the Operating System.

Process Tree in a UNIX system





Termination of a process in UNIX

◆ Exit(status_code)


- Status_code indicate the termination status of the process.
- Kernel does the following after receiving the exit call
 - Close all open files of the process
 - Releases the memory allotted to it
 - Destroy the User area of the process
- It does not destroy the proc structure, this is retained till the parent of Pi destroy it.
- The terminated process is dead but still exists. Hence it is called ZOMBIE process.
- The exit call also sends signal to the parent of Pi which may be ignored by the parent.



Wait statement in UNIX

- Pi can wait for the termination of a child
- `Wait(address(xyz)); // xyz → variable within the address space of Pi.`
- If process Pi has child processes & at least one of them has already terminated, the wait call stores the termination status of the terminated child process into xyz and immediately return with the id of the terminated child process. If more child processes exists their termination status would be made available to Pi only when it repeats the wait call.
- Process Pi is blocked if it has children but none of them has terminated, it will be unblocked when one of the child process terminates. The wait call returns a -1 if Pi has no children.

Example for wait



```
Int main()
{
    int I,saved_status;
    for(I=0;I<3;I++)
    {
        if (fork()==0)
        {
            // Child Process
            exit();
        }
    }
    while(wait(&saved_status)!= -1);    // Loop till all children
    terminates
}
```



Interrupt handling in UNIX

- ◆ Will execute only one interrupt at a time (to avoid race condition).
- ◆ Each interrupt is assigned an interrupt priority level. An interrupt priority level is also assigned to CPU.
- ◆ When an interrupt at priority level '1' arises, it is handled only if $1 > \text{CPU's interrupt priority}$ else kept pending till CPU's interrupt priority level assumes a lower value.

System calls in UNIX

- System calls accepts parameters relevant to its functioning
- When a call occurs these parameters exists on the user stack of the process which issues the system call.
- The call number is expected to be in register 0
- System call handler obtain this number to determine which system functionality being invoked.
- From its internal table it knows the address of the handler for that function.
- Parameters of the system call exists on the user stack of the process making the call.
- Before passing control to the handler for a specific call these parameters are copied from the user stack into some standard places in the user area.



System calls in UNIX Cont...

- A signal can be send to a process/Group of processes using `kill(<pid>,<signum>)` system call. Where `pid` is the id of the process signal to be sent.(sender must know receiver address & receiver must be in the same process tree).
- Pid value 0 \rightarrow signal to be sent to all processes in the same group as the sender process.
- Pid value -1 \rightarrow to reach processes outside the process tree of the sender.



Signal handling in UNIX

- ◆ Oldfunction = signal(<signum>, <function>)
 - Where signal is a function in C library which makes a signal system call. Signum is an integer & function is the function name.
 - The function should be executed on occurrence of signal <signum>.
 - User can specify 0 or 1 instead of function. Where 0 indicates the default action defined in the Kernel is to be executed & 1 → occurrence of signal is to be ignored.



- ◆ Whenever a signal is send to a process, the bit corresponding to the signal is set to 1 in the proc structure of the destination process. The kernel now determines if the signal is being ignored by the destination process. If not it will makes provision to deliver the signal to the process. If the signal is ignored it remains pending and is delivered when the process is ready to accept it. In UNIX signal remains pending if the process to which it is intended is in block state. The signal would be delivered when the process comes out of the blocked state. Signals are delivered when
- ◆ A process returns from a system call, after a process get unblocked, before a process gets blocked.



Interesting signals in UNIX

- ◆ SIGCHLD Child process died or suspended
- ◆ SIGFPE Arithmetic fault
- ◆ SIGILL Illegal instruction
- ◆ SIGINT tty interrupt (Control C)
- ◆ SIGKILL Kill process
- ◆ SIGSEGV Segmentation fault
- ◆ SIGSYS Invalid System call
- ◆ SIGXCPU Exceeds CPU limit
- ◆ SIGXFSZ Exceeds file size limit