# Unit-5

www.mycsvtunotes.in MYcsvtu Notes

# PROCESS STATES AND TRANSITIONS

the lifetime of a process can be conceptually divided into a set of states that describe the process. The following list contains the complete set of process states.

1. The process is executing in user mode.
2. The process is executing in kernel mode.
3. The process is not executing but is ready to run as soon as the kernel schedules it.
4. The process is sleeping and resides in main memory.
5. The process is ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute. Chapter 9 will reconsider this state in a paging system.
6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process. The distinction between this state and state 3 ("ready to run") will be brought out shortly.
8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
9. The process executed the *exit* system call and is in the *zombie* state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.

www.mycsvtunotes.in MYcsvtu Notes

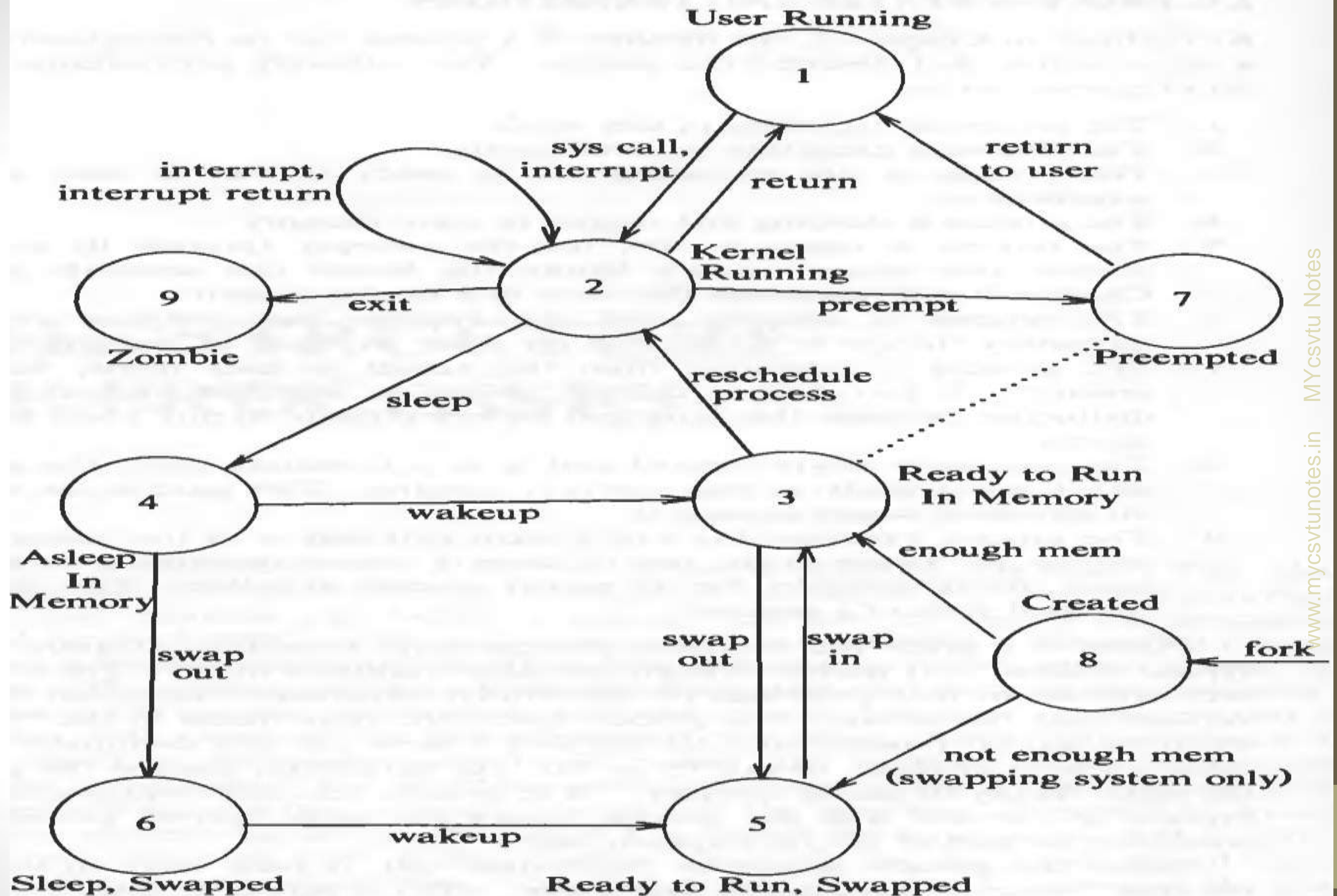


Figure 6.1. Process State Transition Diagram

www.mycsvtunotes.in MYcsvtu Notes

The fields in the process table are the following.

- The *state* field identifies the process state.
- The process table entry contains fields that allow the kernel to locate the process and its *u area* in main memory or in secondary storage. The kernel uses the information to do a *context switch* to the process when the process moves from state "ready to run in memory" to the state "kernel running" or from the state "preempted" to the state "user running." In addition, it uses this information when swapping (or paging) processes to and from main memory (between the two "in memory" states and the two "swapped" states). The process table entry also contains a field that gives the process size, so that the kernel knows how much space to allocate for the process.
- Several user identifiers (user IDs or UIDs) determine various process privileges. For example, the user ID fields delineate the sets of processes that can send signals to each other, as will be explained in the next chapter.
- Process identifiers (process IDs or PIDs) specify the relationship of processes to each other. These ID fields are set up when the process enters the state "created" in the *fork* system call.
- The process table entry contains an event descriptor when the process is in the "sleep" state. This chapter will examine its use in the algorithms for *sleep* and *wakeup*.
- Scheduling parameters allow the kernel to determine the order in which processes move to the states "kernel running" and "user running."
- A signal field enumerates the signals sent to a process but not yet handled (Section 7.2).
- Various timers give process execution time and kernel resource utilization, used for process accounting and for the calculation of process scheduling priority. One field is a user-set timer used to send an alarm signal to a process (Section 8.3).

www.mycsvtunotes.in MYcsvtu Notes

The *u area* contains the following fields that further characterize the process states. Previous chapters have described the last seven fields, which are briefly described again for completeness.

- A pointer to the process table identifies the entry that corresponds to the *u area*.
- The real and effective user IDs determine various privileges allowed the process, such as file access rights (see Section 7.6).
- Timer fields record the time the process (and its descendants) spent executing in user mode and in kernel mode.
- An array indicates how the process wishes to react to signals.
- The control terminal field identifies the "login terminal" associated with the process, if one exists.
- An error field records errors encountered during a system call.
- A return value field contains the result of system calls.
- I/O parameters describe the amount of data to transfer, the address of the source (or target) data array in user space, file offsets for I/O, and so on.
- The current directory and current root describe the file system environment of the process.
- The user file descriptor table records the files the process has *open*.

www.mycsvtunotes.in MYcsvtu Notes

| System Calls Dealing with Memory Management | | | | System Calls Dealing with Synchronization | | | Miscellaneous | |
|---|---|---|---|---|---|---|---|---|
| fork | exec | brk | exit | wait | signal | kill | setpgrp | setuid |
| dupreg<br>attachreg | detachreg<br>allocreg<br>attachreg<br>growreg<br>loadreg<br>mapreg | growreg | detachreg | | | | | |

**Figure 7.1.** Process System Calls and Relation to Other Algorithms

www.mycsvtunotes.in MYcsvtu Notes

# PROCESS CREATION

The only way for a user to create a new process in the UNIX operating system 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. The syntax for the *fork* system call is

pid = fork();

On return from the *fork* system call, 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. Process 0, created internally by the kernel when the system is booted, is the only process not created via *fork*.

The kernel does the following sequence of operations for *fork*.

1. It allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. Since certain portions of a process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory.
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

www.mycsvtunotes.in MYcsvtu Notes

```
algorithm fork
input:  none
output: to parent process, child PID number
        to child process, 0
{
      check for available kernel resources;
      get free proc table slot, unique PID number;
      check that user not running too many processes;
      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       /* executing process is the child process */
      {
            initialize u area timing fields;
            return(0);       /* to user */
      }
}
```

**Figure 7.2. Algorithm for Fork**

www.mycsvtunotes.in MYcsvtu Notes

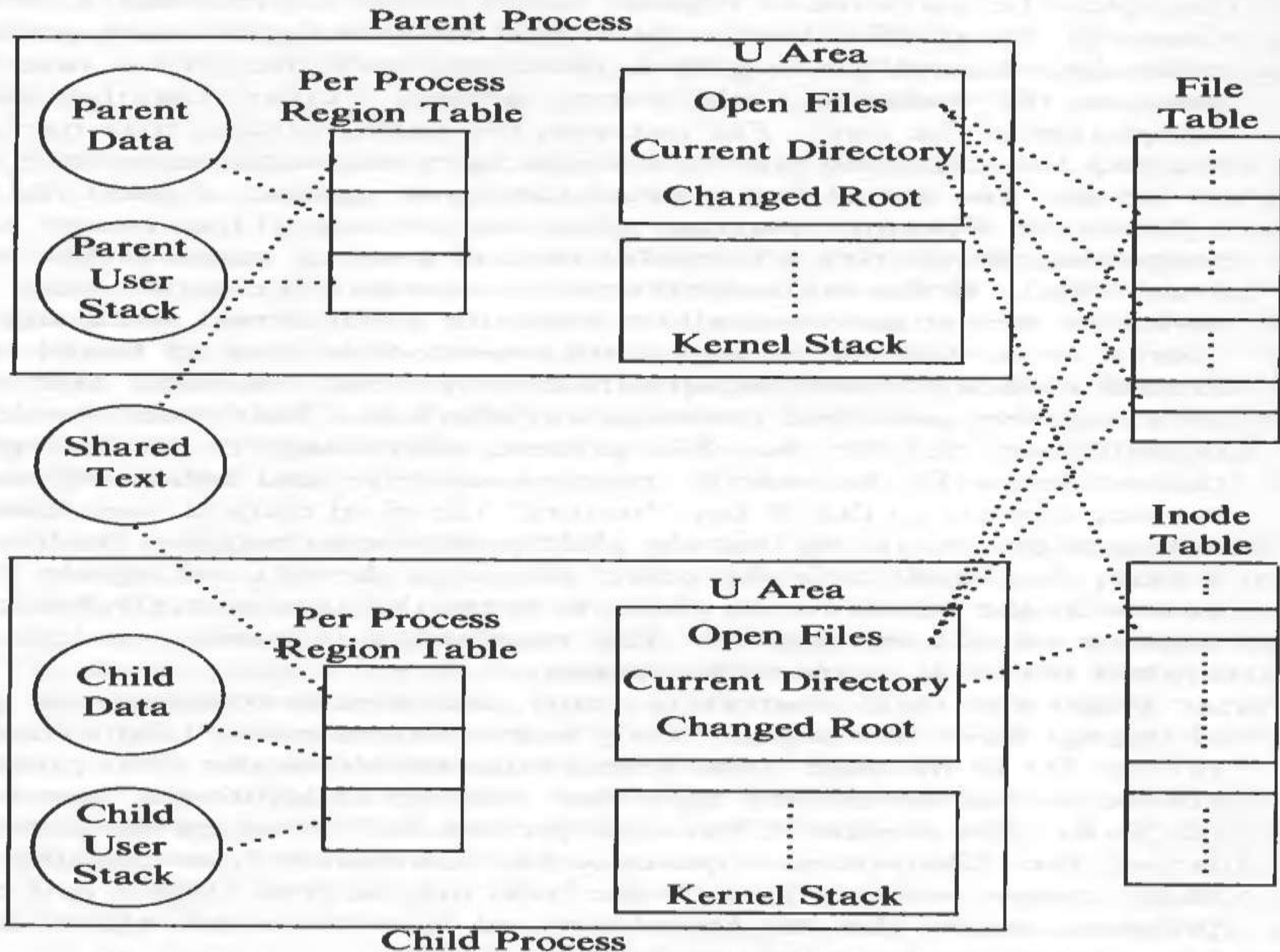


Figure 7.3. Fork Creating a New Process Context

www.mycsvtunotes.in MYcsvtu Notes

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

main(argc, argv)
      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 same code */
      rdwrt();
      exit(0);
}

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

Figure 7.4. Program where Parent and Child Share File Access

www.mycsvtunotes.in MYcsvtu Notes

## 7.3 PROCESS TERMINATION

Processes on a UNIX system terminate by executing the *exit* system call. An *exit*ing process enters the zombie state (recall Figure 6.1), relinquishes its resources, and dismantles its context except for its slot in the process table. The syntax for the call is

```
exit(status);
```

where the value of *status* is returned to the parent process for its examination. Processes may call *exit* explicitly or implicitly at the end of a program: the startup routine linked with all C programs calls *exit* when the program returns from the *main* function, the entry point of all programs. Alternatively, the kernel may invoke *exit* internally for a process on receipt of uncaught signals as discussed above. If so, the value of *status* is the signal number.

The system imposes no time limit on the execution of a process, and processes frequently exist for a long time. For instance, processes 0 (the swapper) and 1 (*init*) exist throughout the lifetime of a system. Other examples are *getty* processes, which monitor a terminal line, waiting for a user to log in, and special-purpose administrative processes.

www.mycsvtunotes.in MYcsvtu Notes

```
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 open 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 of child signal to parent process;
        context switch;
}
```

**Figure 7.14.** Algorithm for Exit

www.mycsvtunotes.in    MYcsvtu Notes

## 7.6 THE USER ID OF A PROCESS

The kernel associates two user IDs with a process, independent of the process ID: the *real user ID* and the *effective user ID* or *setuid* (set user ID). The real user ID identifies the user who is responsible for the running process. The effective user ID is used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes via the *kill* system call. The kernel allows a process to change its effective user ID when it *execs* a *setuid* program or when it invokes the *setuid* system call explicitly.

A *setuid* program is an executable file that has the *setuid* bit set in its permission mode field. When a process *execs* a *setuid* program, the kernel sets the effective user ID fields in the process table and *u area* to the owner ID of the file. To distinguish the two fields, let us call the field in the process table the *saved* user ID. An example illustrates the difference between the two fields.

The syntax for the *setuid* system call is

```
setuid(uid)
```

where *uid* is the new user ID, and its result depends on the current value of the effective user ID. If the effective user ID of the calling process is superuser, the kernel resets the real and effective user ID fields in the process table and *u area* to *uid*. If the effective user ID of the calling process is not superuser, the kernel resets the effective user ID in the *u area* to *uid* if *uid* has the value of the real user ID or if it has the value of the saved user ID. Otherwise, the system call returns an error. Generally, a process inherits its real and effective user IDs from its parent during the *fork* system call and maintains their values across *exec* system calls.

www.mycsvtunotes.in MYcsvtu Notes

## 7.7 CHANGING THE SIZE OF A PROCESS

A process may increase or decrease the size of its data region by using the *brk* system call. The syntax for the *brk* system call is

```
brk(endds);
```

where *endds* becomes the value of the highest virtual address of the data region of the process (called its *break* value). Alternatively, a user can call

```
oldendds = sbrk(increment);
```

where *increment* changes the current break value by the specified number of bytes, and *oldendds* is the break value before the call.

www.mycsvtunotes.in MYcsvtu Notes

```
algorithm brk
input:   new break value
output: old break value
{
        lock process data region;
        if (region size increasing)
                if (new region size is illegal)
                {
                        unlock data region;
                        return(error);
                }
        change region size (algorithm growreg);
        zero out addresses in new data space;
        unlock process data region;
}
```

**Figure 7.26.** Algorithm for Brk

www.mycsvtunotes.in MYcsvtu Notes

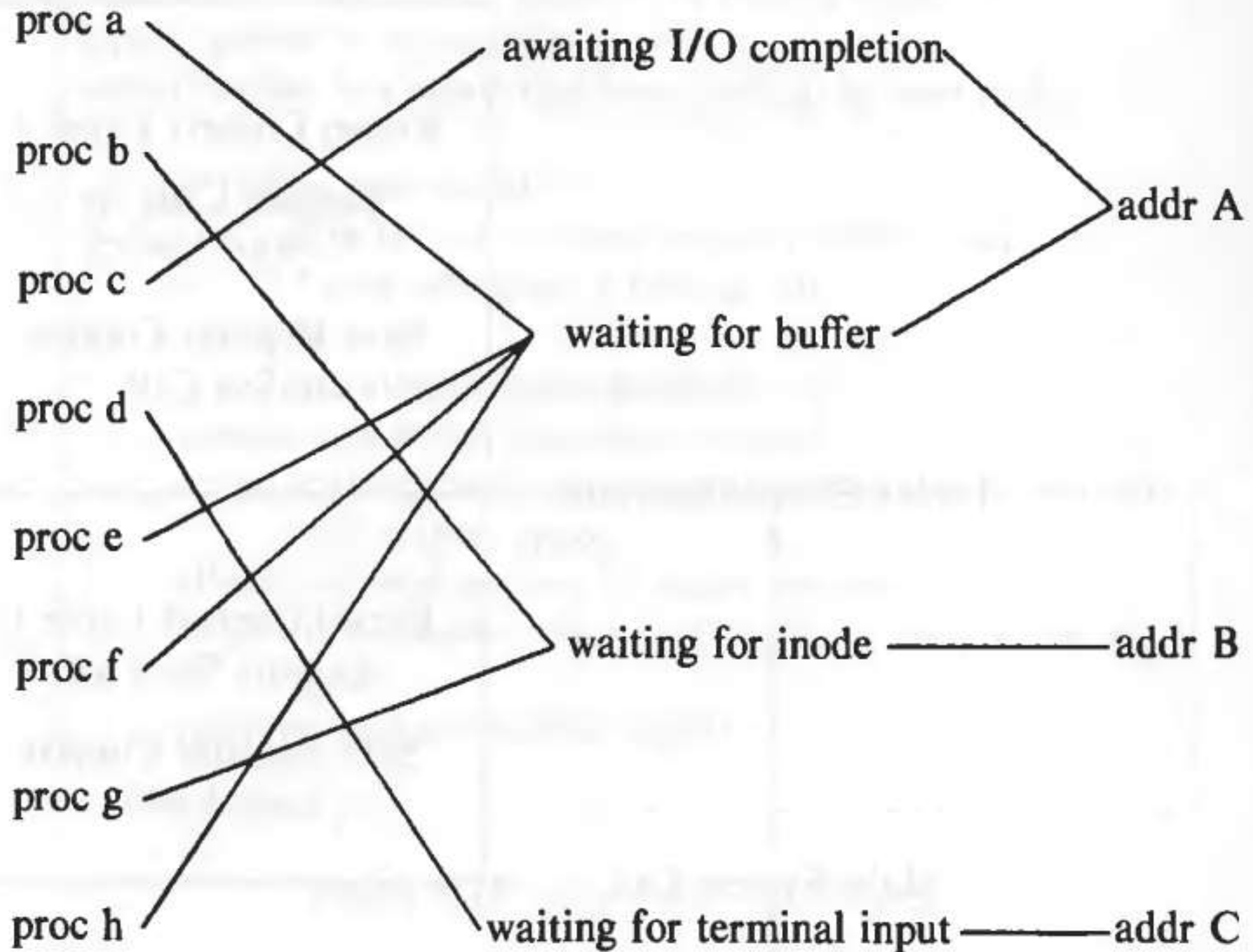


Figure 6.30. Processes Sleeping on Events and Events Mapping into Addresses

www.mycsvtunotes.in MYcsvtu Notes

```
algorithm sleep
input:   (1) sleep address
         (2) priority
output: 1 if process awakened as a result of a signal that process catches,
        longjump algorithm if process awakened as a result of a signal
                              that it does not catch,
        0 otherwise;
{
      raise processor execution level to block all interrupts;
      set process state to sleep;
      put process on sleep hash queue, based on sleep address;
      save sleep address in process table slot;
      set process priority level to input priority;
      if (process sleep is NOT interruptible)
      {
            do context switch;
            /* process resumes execution here when it wakes up */
            reset processor priority level to allow interrupts as when
                  process went to sleep;
            return(0);
      }

      /* here, process sleep is interruptible by signals */
      if (no signal pending against process)
      {
            do context switch;
            /* process resumes execution here when it wakes up */
            if (no signal pending against process)
            {
                  reset processor priority level to what it was when
                        process went to sleep;
                  return(0);
            }
      }
      remove process from sleep hash queue, if still there;

      reset processor priority level to what it was when process went to sleep;
      if (process sleep priority set to catch signals)
            return(1)
      do longjmp algorithm;
}
```

**Figure 6.31. Sleep Algorithm**

www.mycsvtunotes.in MYcsvtu Notes

```
algorithm wakeup                    /* wake up a sleeping process */
input:   sleep address
output: none
{
      raise processor execution level to block all interrupts;
      find sleep hash queue for sleep address;
      for (every process asleep on sleep address)
      {
            remove process from hash queue;
            mark process state "ready to run";
            put process on scheduler list of processes ready to run;
            clear field in process table entry for sleep address;
            if (process not loaded in memory)
                  wake up swapper process (0);
            else if (awakened process is more elligible to run than
                              currently running process)
                  set scheduler flag;
      }
      restore processor execution level to original level;
}
```

**Figure 6.32.** Algorithm for Wakeup

www.mycsvtunotes.in MYcsvtu Notes