Processes and Threads

경희대학교 컴퓨터공학과

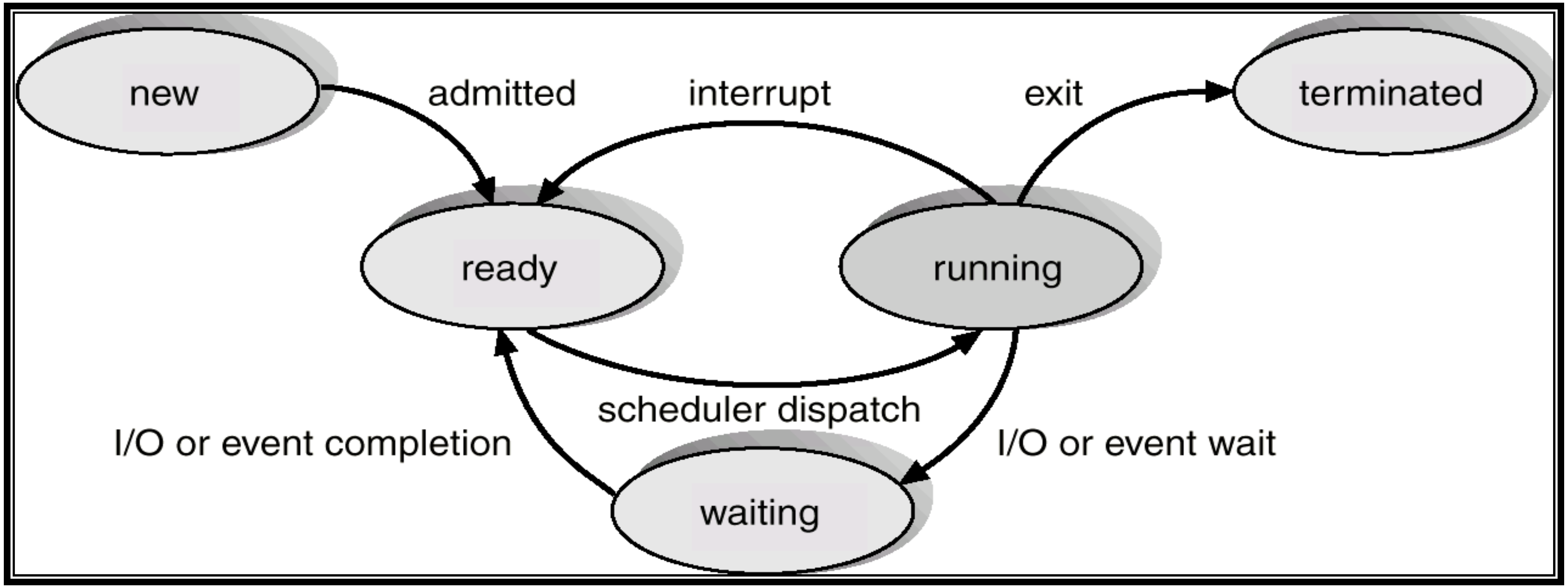
조 진 성

Program vs. Process

- Program
  - ✓ Executable file on a disk
  - ✓ Loaded into memory and executed by the kernel

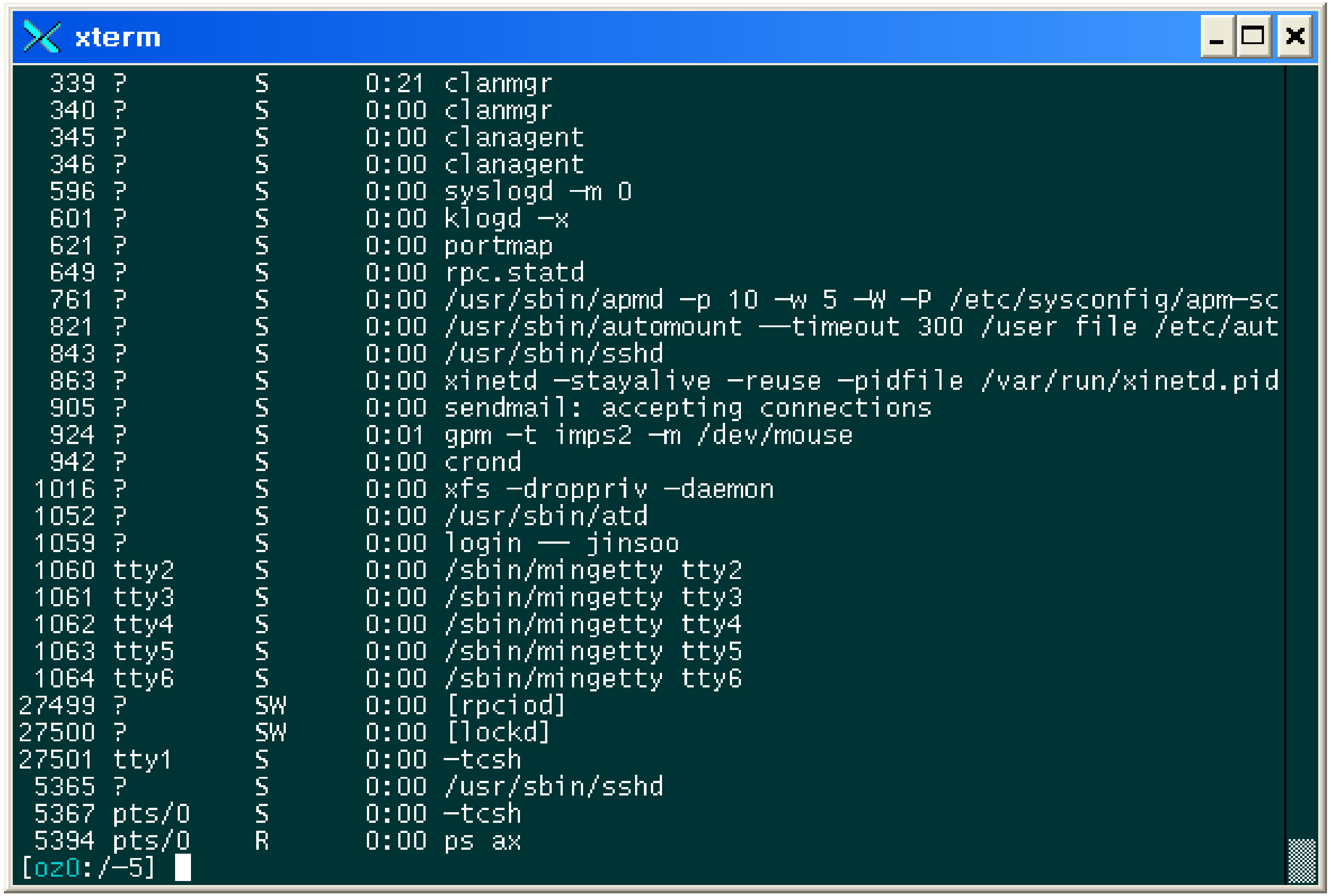
- Process
  - ✓ Executing instance of a program
  - ✓ The basic unit of execution and scheduling
  - ✓ A process is named using its process ID (PID)
  - ✓ Other IDs associated with a process
    - real user ID
    - real group ID
    - effective user ID
    - effective group ID
    - saved set-user-ID
    - saved set-group-ID

Process State



Process State (Cont'd)

- "ps" command



```
  339 ?        S      0:21 clanmgr
  340 ?        S      0:00 clanmgr
  345 ?        S      0:00 clanagent
  346 ?        S      0:00 clanagent
  596 ?        S      0:00 syslogd -m 0
  601 ?        S      0:00 klogd -x
  621 ?        S      0:00 portmap
  649 ?        S      0:00 rpc.statd
  761 ?        S      0:00 /usr/sbin/apmd -p 10 -w 5 -W -P /etc/sysconfig/apm-sc
  821 ?        S      0:00 /usr/sbin/automount --timeout 300 /user file /etc/aut
  843 ?        S      0:00 /usr/sbin/sshd
  863 ?        S      0:00 xinetd -stayalive -reuse -pidfile /var/run/xinetd.pid
  905 ?        S      0:00 sendmail: accepting connections
  924 ?        S      0:01 gpm -t imps2 -m /dev/mouse
  942 ?        S      0:00 crond
 1016 ?        S      0:00 xfs -droppriv -daemon
 1052 ?        S      0:00 /usr/sbin/atd
 1059 ?        S      0:00 login -- jinsoo
 1060 tty2     S      0:00 /sbin/mingetty tty2
 1061 tty3     S      0:00 /sbin/mingetty tty3
 1062 tty4     S      0:00 /sbin/mingetty tty4
 1063 tty5     S      0:00 /sbin/mingetty tty5
 1064 tty6     S      0:00 /sbin/mingetty tty6
27499 ?        SW     0:00 [rpciod]
27500 ?        SW     0:00 [lockd]
27501 tty1     S      0:00 -tcsh
 5365 ?        S      0:00 /usr/sbin/sshd
 5367 pts/0    S      0:00 -tcsh
 5394 pts/0    R      0:00 ps ax
[oz0:/-5]
```

R: Runnable
S: Sleeping
T: Traced or Stopped
D: Uninterruptible Sleep
Z: Zombie

W: No resident pages
<: High-priority task
N: Low-priority task
L: Has pages locked into memory

IDs Associated with a Process

- Get various IDs associated with a process
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <unistd.h>`
  - ✓ `pid_t getpid(void);`
  - ✓ return: process ID of calling process
  - ✓ `pid_t getppid(void);`
  - ✓ return: parent process ID of calling process
  - ✓ `uid_t getuid(void);`
  - ✓ return: real user ID of calling process
  - ✓ `uid_t geteuid(void);`
  - ✓ return: effective user ID of calling process
  - ✓ `gid_t getgid(void);`
  - ✓ return: group ID of calling process
  - ✓ `gid_t getegid(void);`
  - ✓ return: effective group ID of calling process

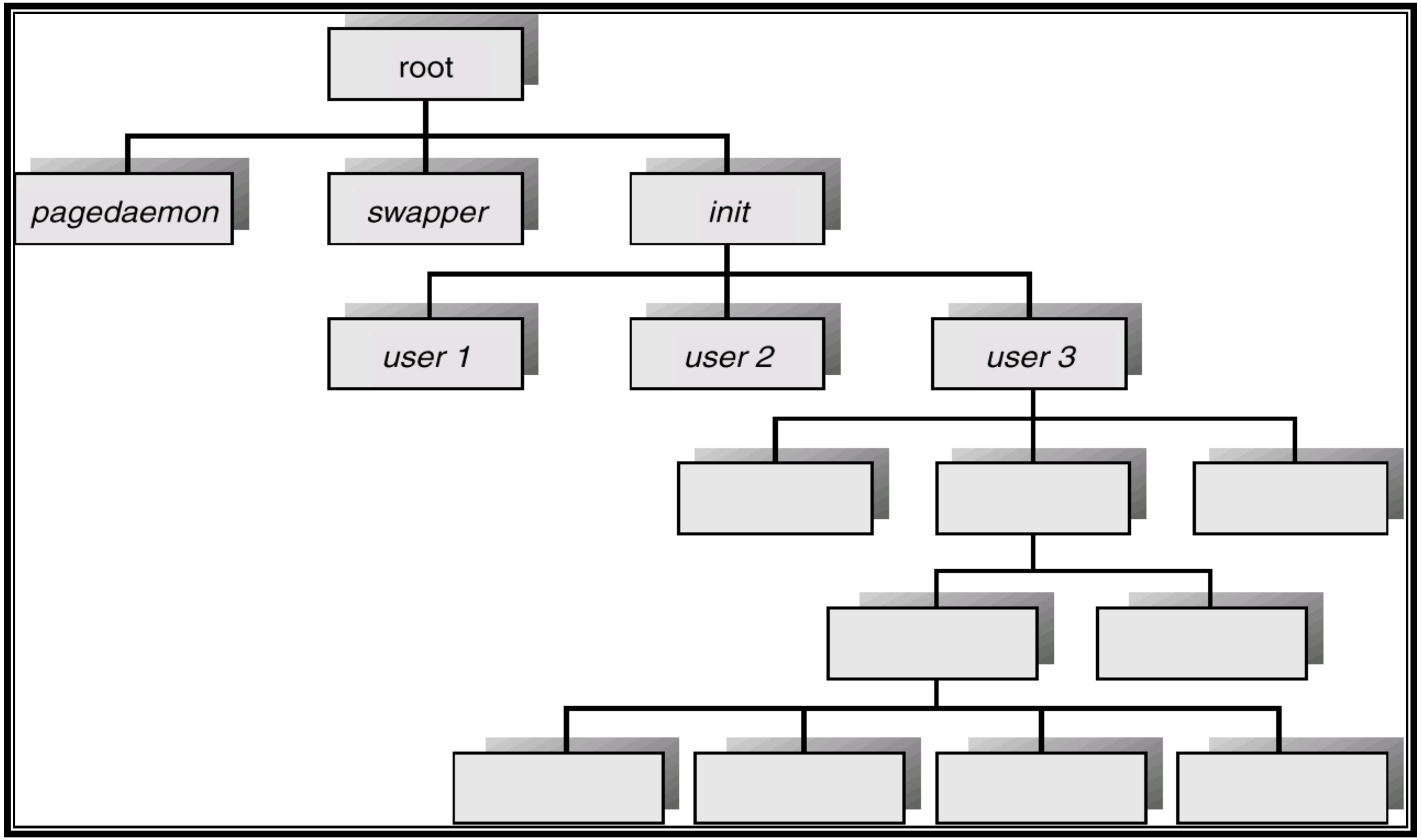
Create a New Process

- fork: the only way a new process is created in Linux
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <unistd.h>`
  - ✓ `pid_t fork(void);`
  - ✓ return: 0 in child, process ID of child in parent, –1 on error

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
main()
{
    int pid;
    if ((pid = fork()) == 0)
        /* child */
        printf("I am %d. My parent is %d\n", getpid(), getppid());
    else
        /* parent */
        printf("I am %d. My child is %d\n", getpid(), pid);
}
```

Create a New Process (Cont'd)

- Process tree in Linux



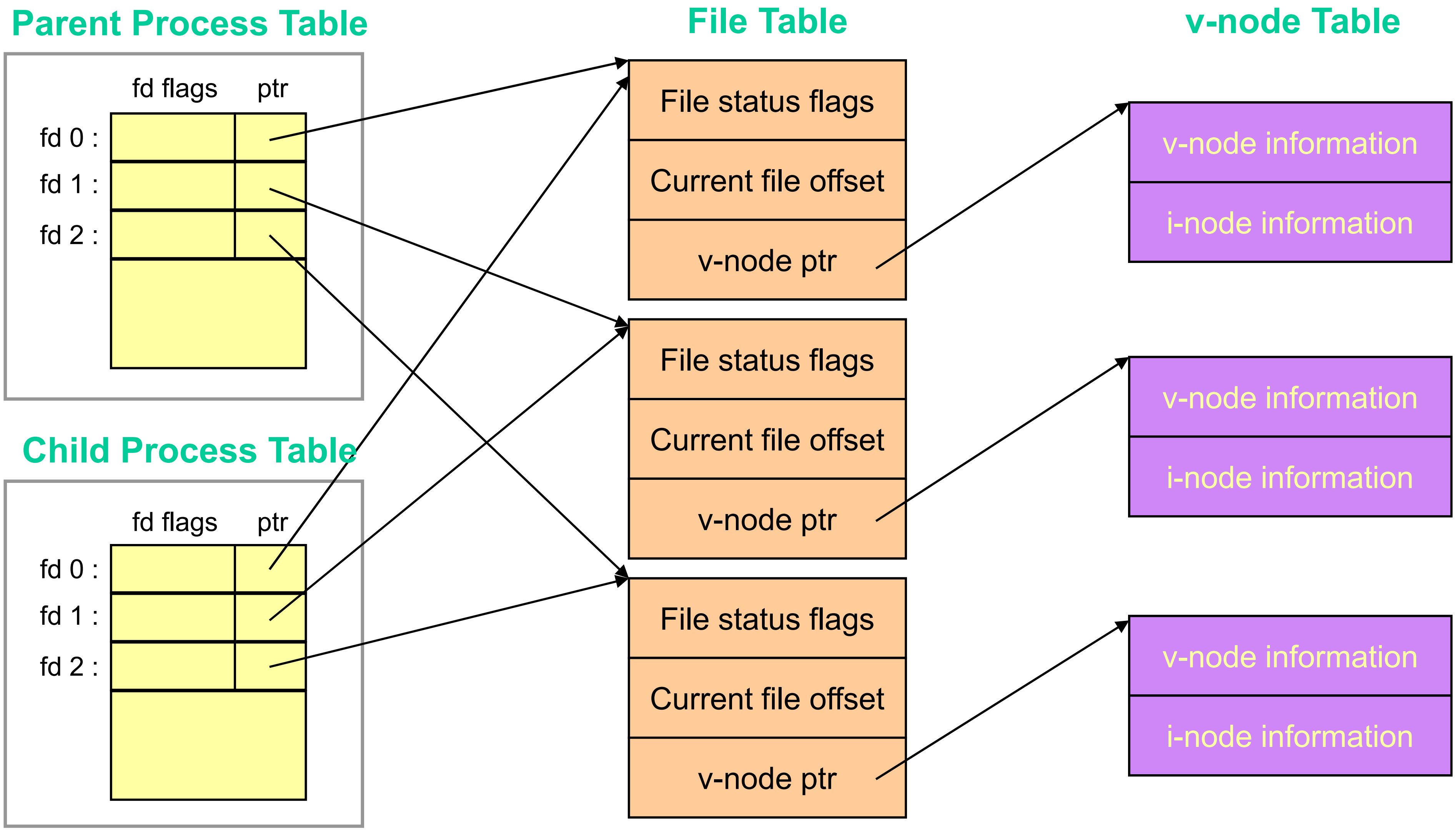
Create a New Process (Cont'd)

- Why fork() ? → Very useful when the child…
  - ✓ is cooperating with the parent
  - ✓ relies upon the parent's data to accomplish its task
  - ✓ Example: Web server

```
while (1) {
     int sock = accept();
     if ((pid = fork()) == 0) {
          /* Handle client request */
     } else {
          /* Close socket */
     }
}
```

Create a New Process (Cont'd)

- Sharing of open files between parent and child after **`fork`**



Exercise

- `fork` example

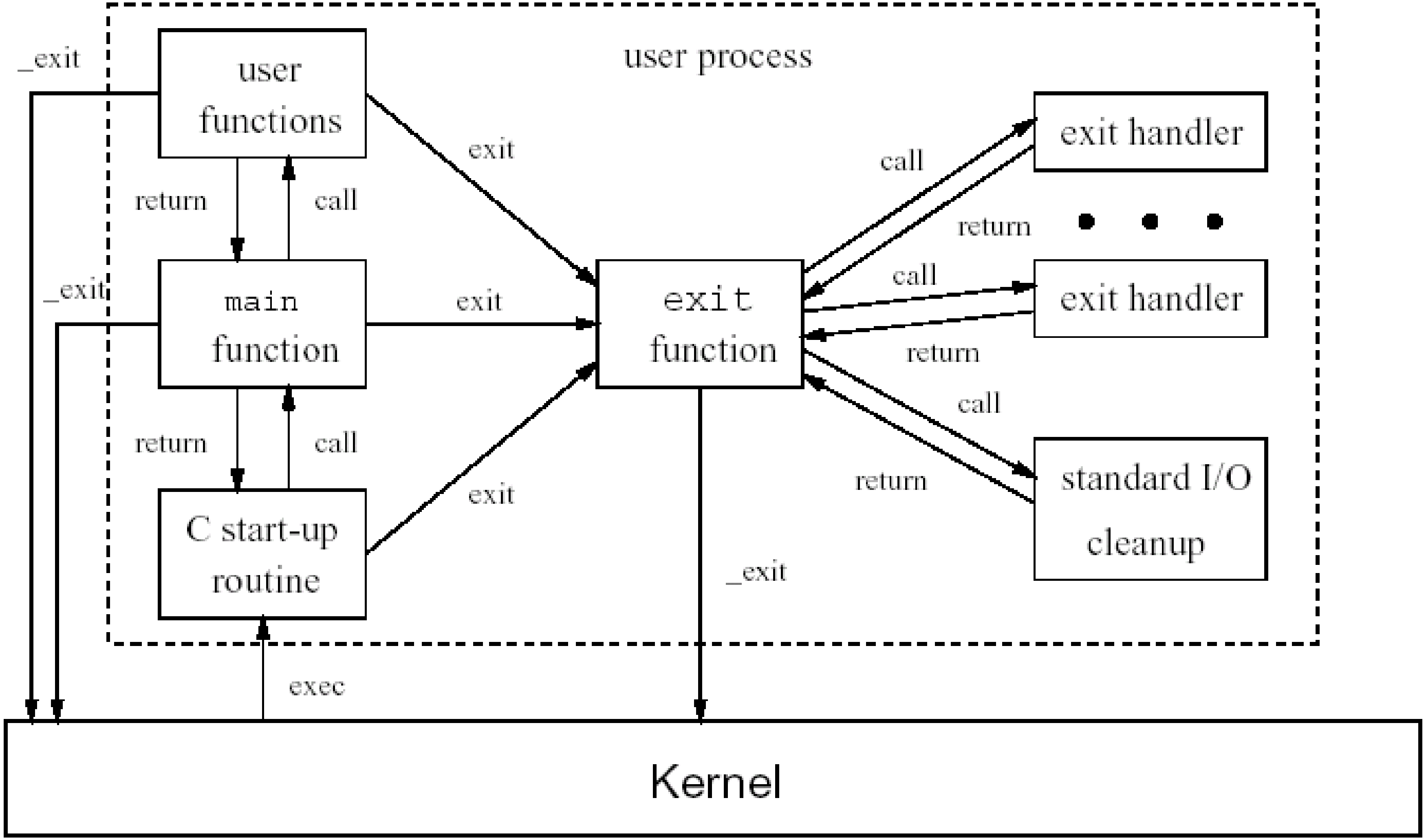
```
$ gcc –o fork fork.c (or make fork)
$ ./fork
```

Terminate a Process

- Normal termination
  - ✓ return from **`main()`**
  - ✓ calling **`exit()`**
  - ✓ calling **`_exit()`**

- Abnormal termination
  - ✓ calling **`abort()`**
  - ✓ terminated by a signal

Start and Termination of a C Program



Terminate a Process

- exit
  - `#include <stdlib.h>`
  - `void exit(int status);`
  - return: 0 if OK, nonzero on error

  - `#include <unistd.h>`
  - `void _exit(int status);`
  - return: 0 if OK, nonzero on erro

- Register an exit handler
  - `#include <stdlib.h>`
  - `int atexit(void (*func)(void));`
  - return: 0 if OK, nonzero on error

Exercise

- `atexit` example

```
$ gcc –o exit exit.c (or make exit)
$ ./exit
```

Wait for Process Termination

- wait
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <sys/wait.h>`
  - ✓ `pid_t wait(int *statloc);`
  - ✓ `pid_t waitpid(pid_t pid, int *statloc, int options);`
  - ✓ return: process ID if OK, 0 (see below) or –1 on error
    - With **`WNOHANG`** option, **`waitpid`** will not block if a child specified by **`pid`** is not immediately available. In this case, the return value is 0
  - ✓ The calling process will
    - block (if all of its children are still running)
    - return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
    - return immediately with an error (if it doesn't have any child processes)

- Cf) **`SIGCHLD`** signal
  - ✓ asynchronous event

Exercise

- `wait` example

```
$ gcc –o wait wait.c (or make wait)
$ ./wait
```

- A program with race condition

```
$ gcc –o race race.c (or make race)
$ ./race
```

- Modification to avoid race condition using `wait` system call

```
$ gcc –o worace worace.c (or make worace)
$ ./worace
```

Execute Another Program in a Program

- exec
  - `#include <unistd.h>`
  - `int execl(char *pathname, char *arg0, ... /* (char *)0 */ );`
  - `int execv(char *pathname, char *argv[]);`

  -
    ```
    int execle(char *pathname, char *arg0, ... /* (char *)0,
                    char *envp[] */ );
    ```
  - `int execve(char *pathname, char *argv[], char *envp[]);`
  - `int execlp(char *filename, char *arg0, ... /* (char *)0 */ );`
  - `int execvp(char *filename, char *argv[]);`
  - return: –1 on error, no return on success

Execute a Command String in a Program

- system
  - ✓ `#include <stdlib.h>`
  - ✓ `int system(char *cmdstring);`
  - ✓ return: termination status of the shell if OK, –1 on error
  - ✓ **`system`** is implemented by calling **`fork`**, **`exec`**, and **`waitpid`**

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  system("ls -al");
  system("date");
  system("who");
}
```

Exercise

- Access environment variables

```
$ gcc –o env env.c (or make env)
$ ./env
```

- `exec` example

```
$ gcc –o exec exec.c (or make exec)
$ ./exec
```

- `system` example

```
$ gcc –o system system.c (or make system)
$ ./system
```

Process vs. Thread

- Why thread ?
  - ✓ Web server example using process
    - Using fork() to create new processes to handle requests in parallel is overkill for such a simple task

```
while (1) {
     int sock = accept();
     if ((pid = fork()) == 0) {
          /* Handle client request */
     } else {
          /* Close socket */
     }
}
```

Process vs. Thread (Cont'd)

- Why thread ? (Cont'd)
  - ✓ Web server example using thread
    - We can create a new thread for each request

```
webserver ()
{
     while (1) {
          int sock = accept();
          thread_fork(handle_request, sock);
     }
}

handle_request (int sock)
{
     /* Process request */
     close (sock);
}
```

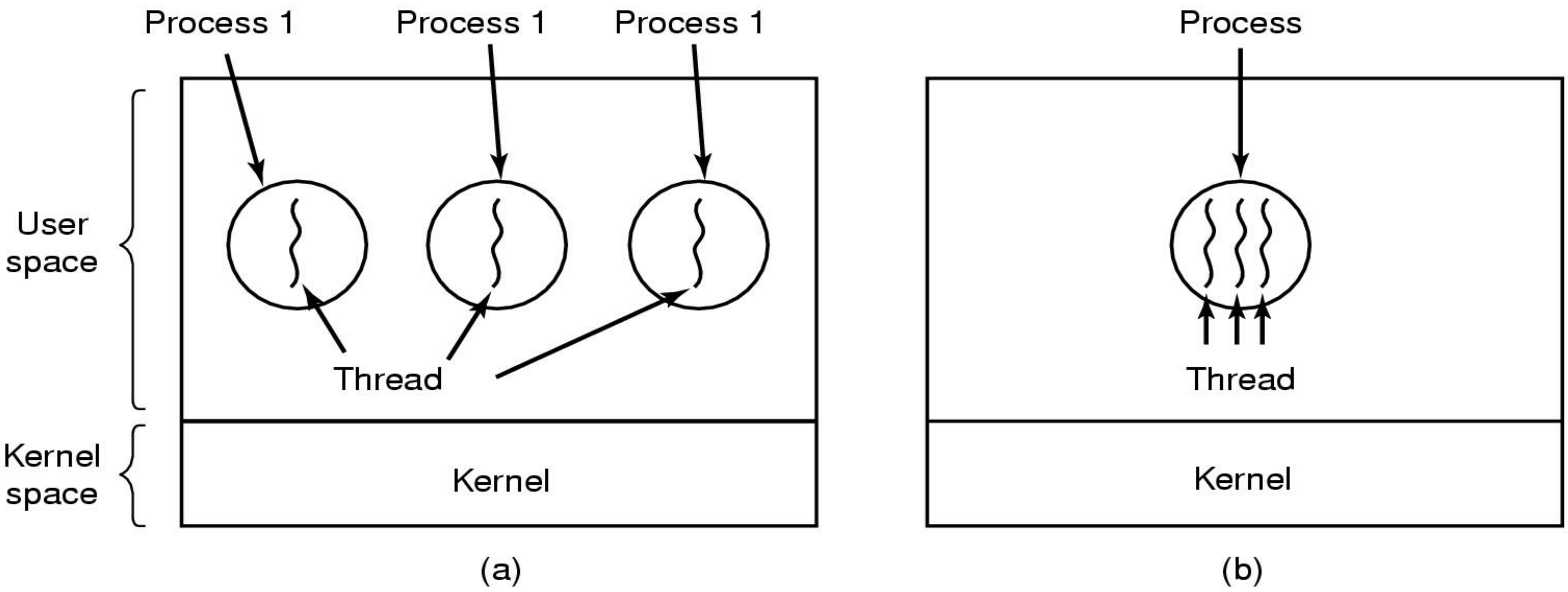
Process vs. Thread (Cont'd)

- Why thread ? (Cont'd)
  - ✓ Responsiveness
  - ✓ Resource Sharing
  - ✓ Economy
  - ✓ Utilization of MP Architectures

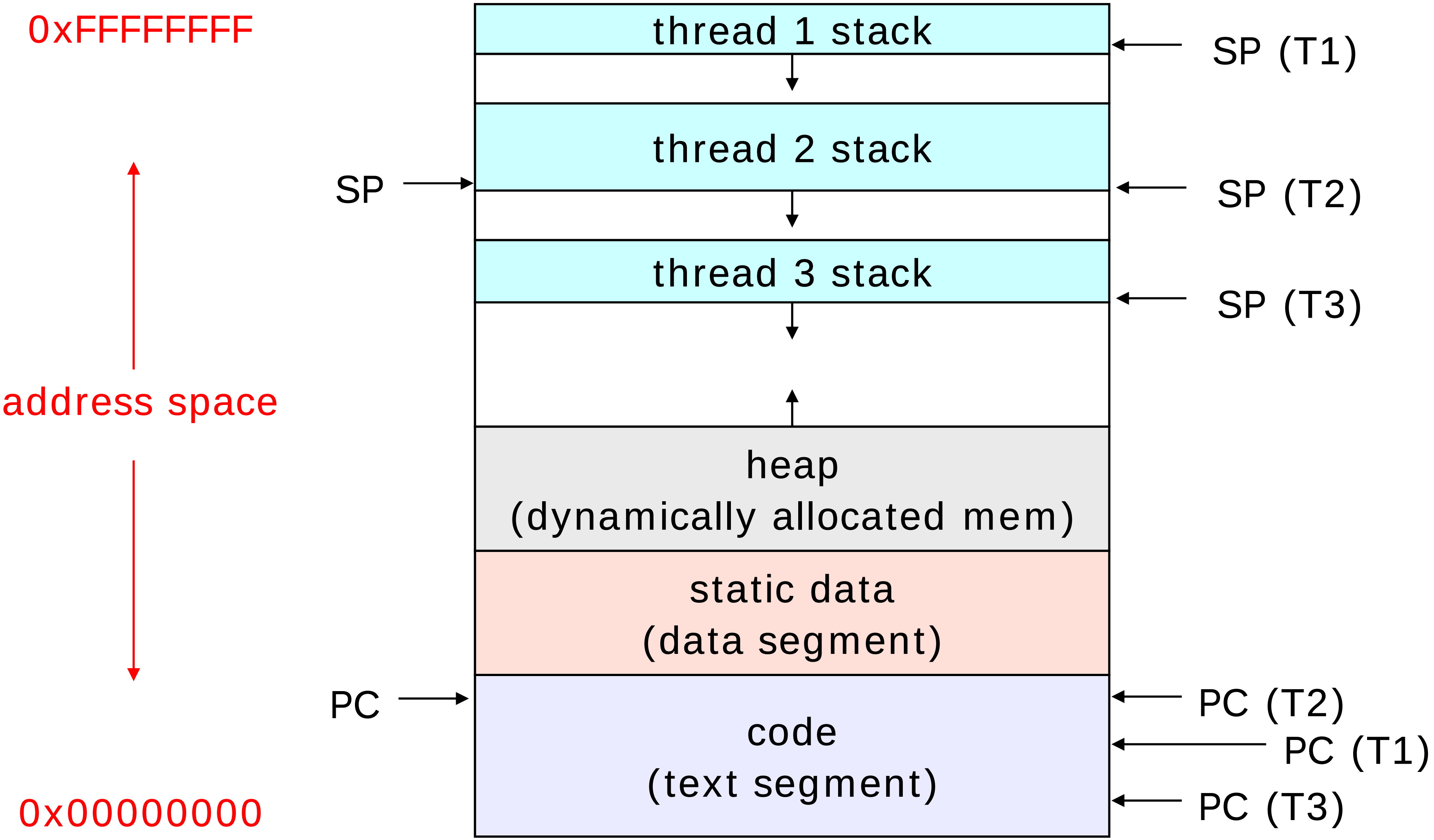
Thread Concept

- Separate the concept of a process from its execution state
  - ✓ Process: address space, resources, other general process attributes (e.g., privileges)
  - ✓ Execution state: PC, SP, registers, etc.
  - ✓ This execution state is usually called
    - a thread of control,
    - a thread, or
    - a lightweight process (LWP)

Thread Concept (Cont'd)

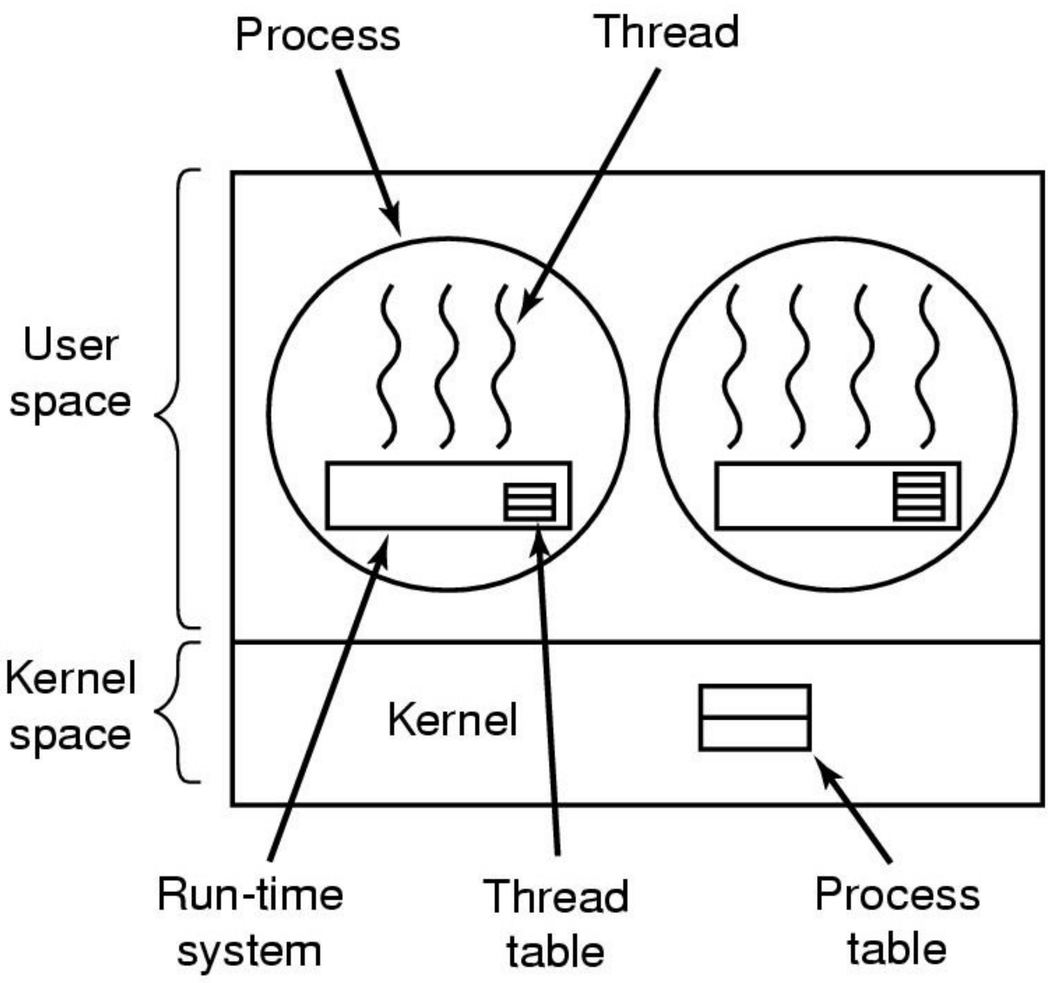


Address Space with Threads

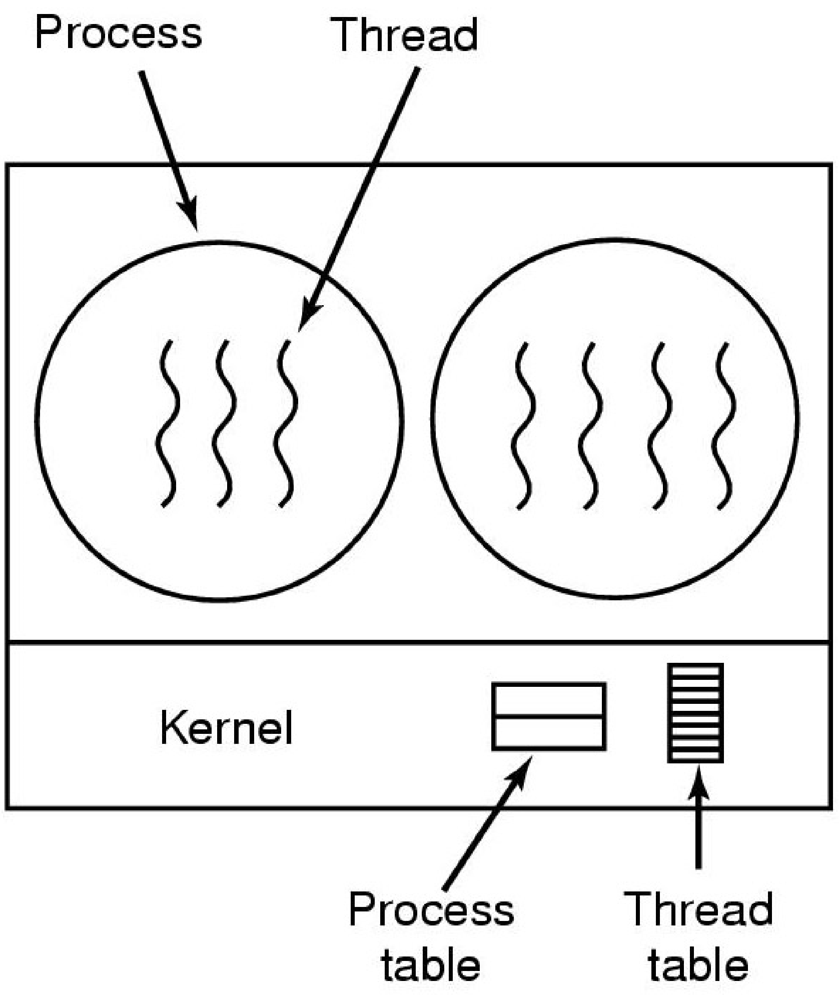


Thread Implementation

User-level Threads



Kernel-level Threads



Thread Implementation (Cont'd)

- User-level threads
  - ✓ The user-level threads library implements thread operations
  - ✓ They are small and fast
  - ✓ User-level threads are invisible to the OS
  - ✓ OS may make poor decisions
    - e.g. blocking I/O
  - ✓ Thread scheduling
    - Non-preemptive scheduling: yield()
    - Preemptive scheduling: timer through signal

- Kernel-level threads
  - ✓ All thread operations are implemented in the kernel
  - ✓ The OS schedules all of the threads in a system
  - ✓ Kernel threads are cheaper than processes
  - ✓ They can still be too expensive

Thread Implementation (Cont'd)

- Pthreads
  - ✓ A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - ✓ API specifies behavior of the thread library, implementation is up to development of the library
  - ✓ Common in UNIX operating systems
  - ✓ Link with **`–lpthread`** option
- Solaris implementation
  - ✓ via Light-Weight Process (LWP)
  - ✓ Kernel-level implementation (including user-level implementation)
- Linux implementation
  - ✓ Kernel-level implementation, but....
    - a modified process(or task) per thread
  - ✓ System call clone( ) for thread creation
  - ✓ NGPT (Next Generation POSIX Threading) by IBM

Pthread Libraries for Thread Control

- Create a thread
  - ✓ `#include <pthread.h>`
  - ✓
    ```
    int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                    void *(start_routine)(void *), void *arg);
    ```
  - ✓ return: 0 if OK, nonzero on error

- Terminate a thread
  - ✓ `#include <pthread.h>`
  - ✓ `void pthread_exit(void *retval);`

- Wait for termination of another thread
  - ✓ `#include <pthread.h>`
  - ✓ `int pthread_join(pthread_t tid, void **tread_return);`
  - ✓ return: 0 if OK, nonzero on error

Exercise

- pthread example

```
$ gcc –o thread thread.c –lpthread (or make thread)
$ ./thread
```

- Command-line processor: iteration version using one process

```
$ gcc –o cmd_i cmd_i.c (or make cmd_i)
$ ./cmd_i
CMD> doit
Doing doit
Done
CMD> quit
```

- Command-line processor: a process per command

```
$ gcc –o cmd_p cmd_p.c (or make cmd_p)
$ ./cmd_p
```

- Command-line processor: a thread per command

```
$ gcc –o cmd_t cmd_t.c –lpthread (or make cmd_t)
$ ./cmd_t
```

Summary

- System calls in Linux for processes and threads
  - ✓ `getpid, getppid, getuid, geteuid, getgid, getegid`
  - ✓ `fork`
  - ✓ `exit, atexit`
  - ✓ `wait, waitpid`
  - ✓ `execl, execv, execle, execve, execlp, execvp`
  - ✓ `system`
  - ✓ `pthread_create, pthread_exit, pthread_join`