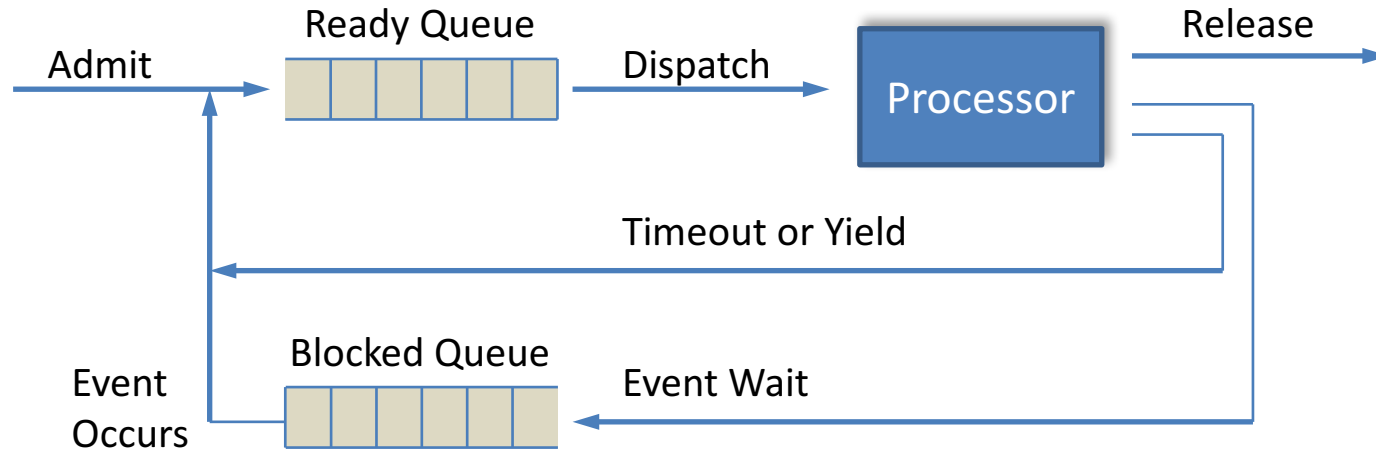


Processes

CS3026 Operating Systems

Lecture 05

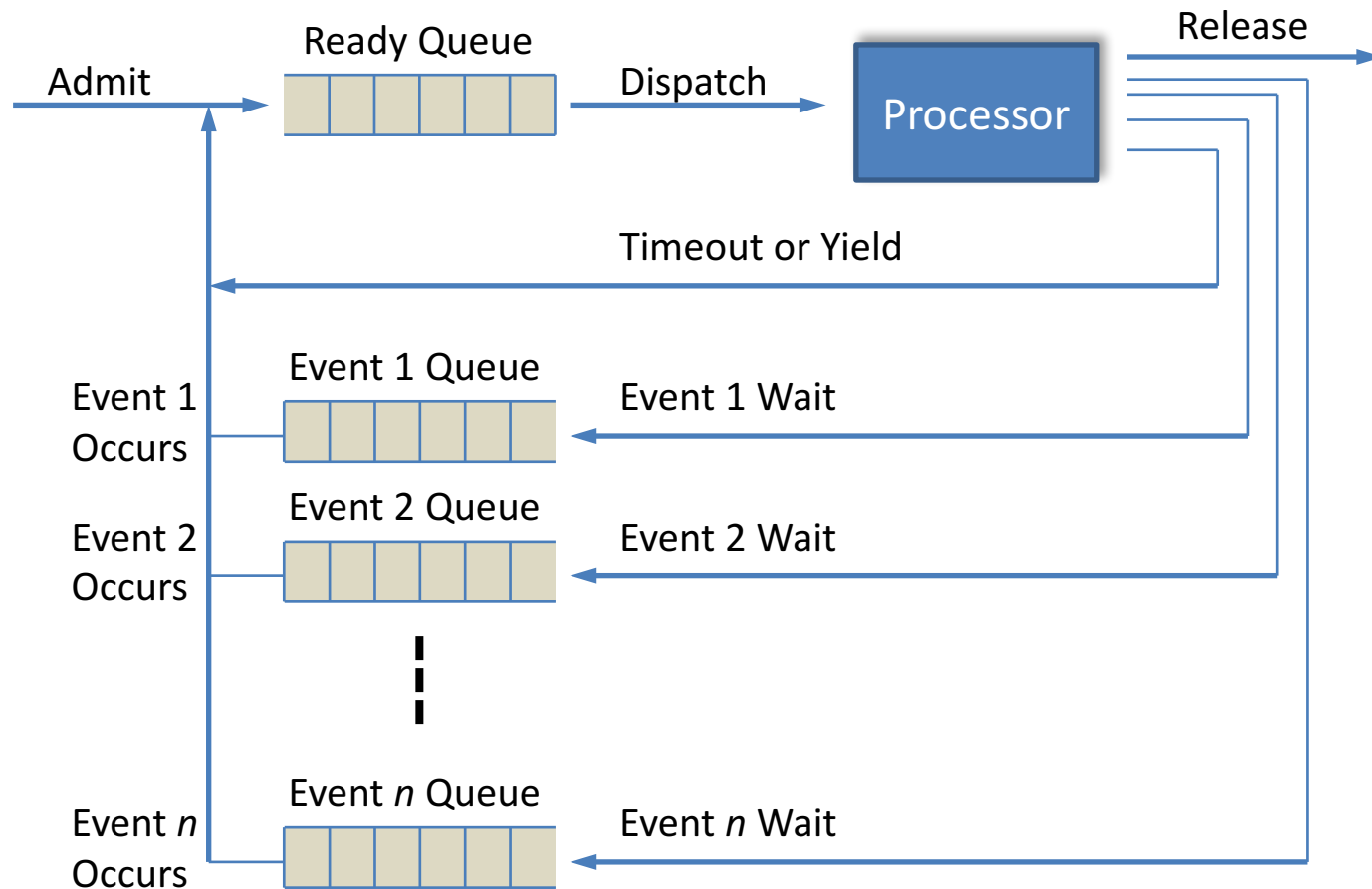
Dispatcher



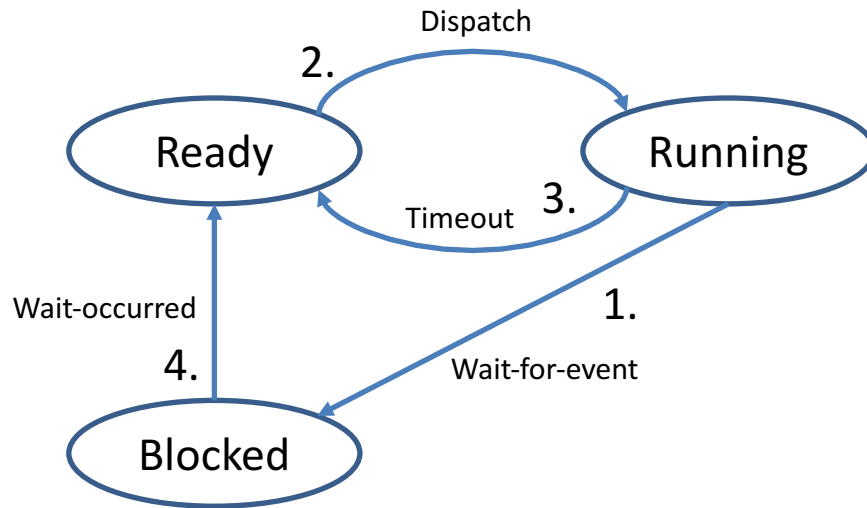
- Implementation:
 - Using one “Ready” and one “Blocked” queue
 - Weakness:
 - When a particular event occurs, ALL processes waiting for this event have to be transferred from the “Blocked” queue to the “Ready” queue
 - Operating system has to look through all the entries in the Blocked queue to select the right processes for transfer

Process Management

- Multiple Event Queues



Process Execution (Dispatch)

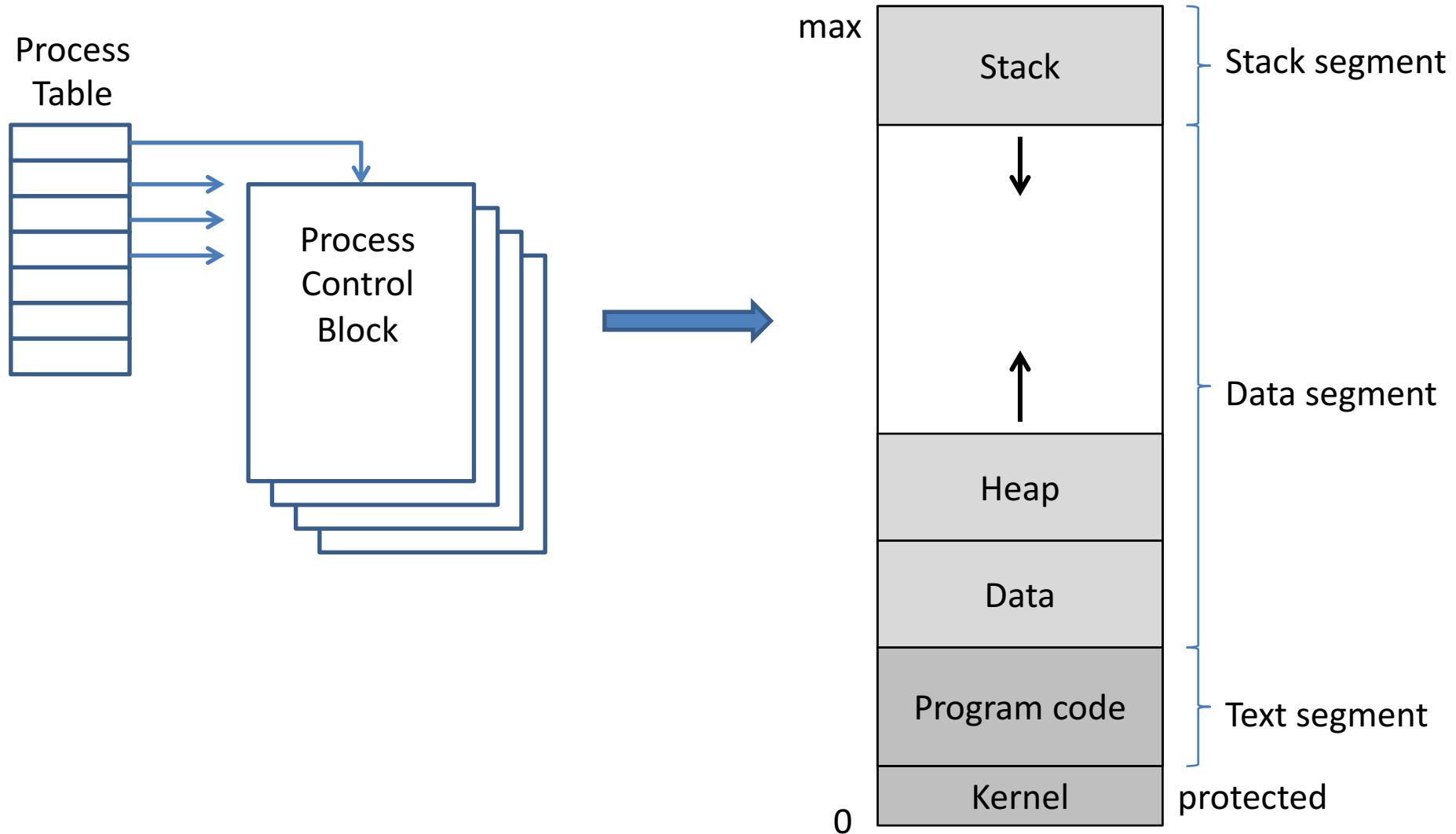


1. Process blocked for I/O
2. Dispatcher schedules another process
3. Dispatcher interrupts process because its time slice expired
4. Input becomes available, blocked process made ready

- We can distinguish three basic process states during execution
 - Running: actually using the CPU
 - Ready: being runnable, temporarily stopped (time-out) to let another process execute
 - Blocked/Waiting: unable to run until some external event happens, such as I/O completion

Process Creation

Process Creation in Unix



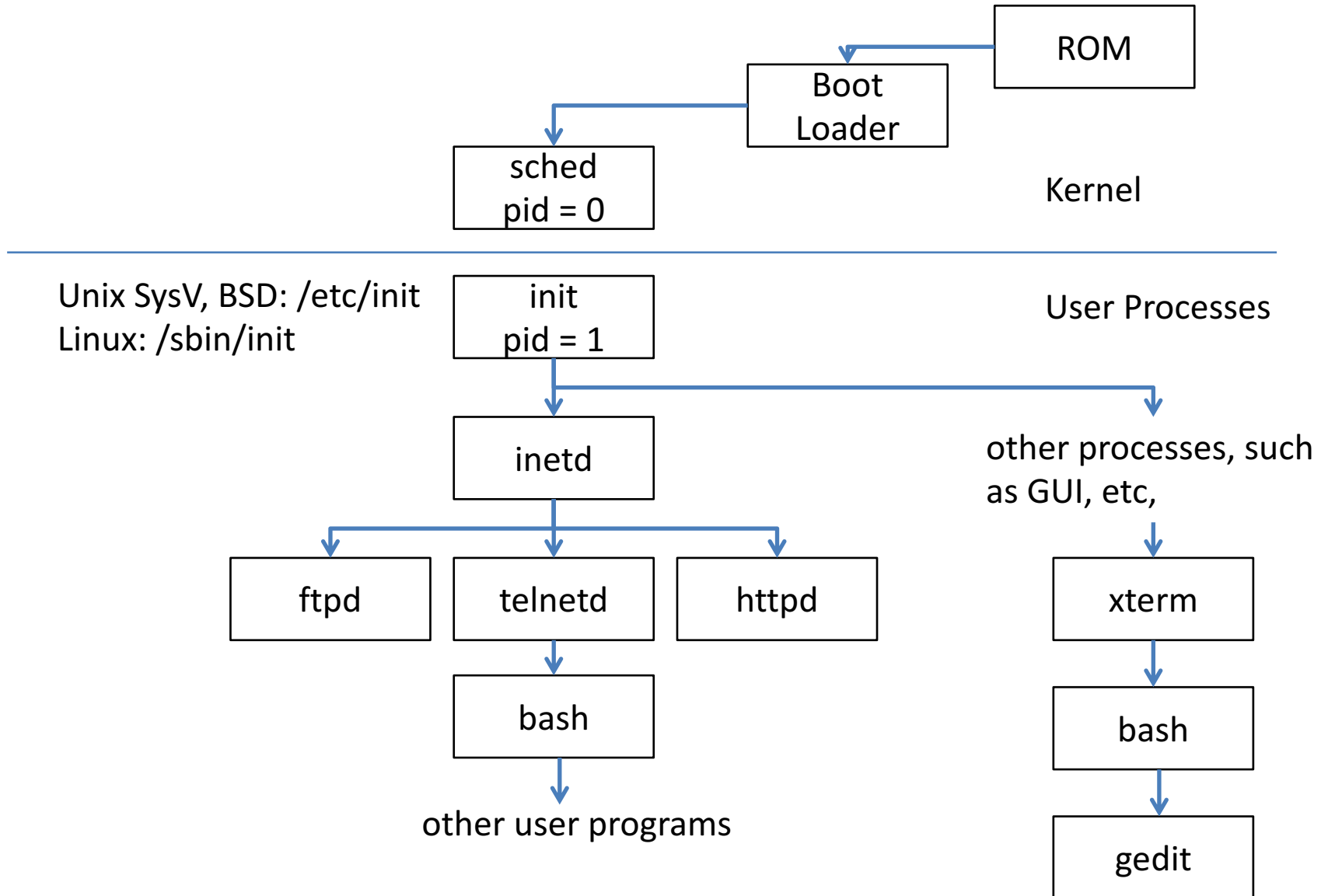
Process Creation

- State “New”: process is created, but not yet admitted to the pool of executable processes by the operating system
 - Process Control Block created
 - Program may not be loaded yet into main memory
- Assign unique process identifier to the new process
- Initialise the Process Control Block
- Allocate memory for process image (program, data, stack)
- Load program
- Add process to the Ready queue

Process Creation in Unix

- Unix has a strict process hierarchy
 - System boot
 - When a system is initialised, several processes are started
 - Under Unix, these are background processes, called “daemons”, such as sched (pid 0) and init (pid 1), and others such as email server, logon server, inetd etc.
- Principal process creation mechanism
 - Use of the **fork()** system call
 - a parent process creates a new child process
 - Child process is a clone of the parent process
 - Use the **exec()** system call to replace the content of the clone with a new program

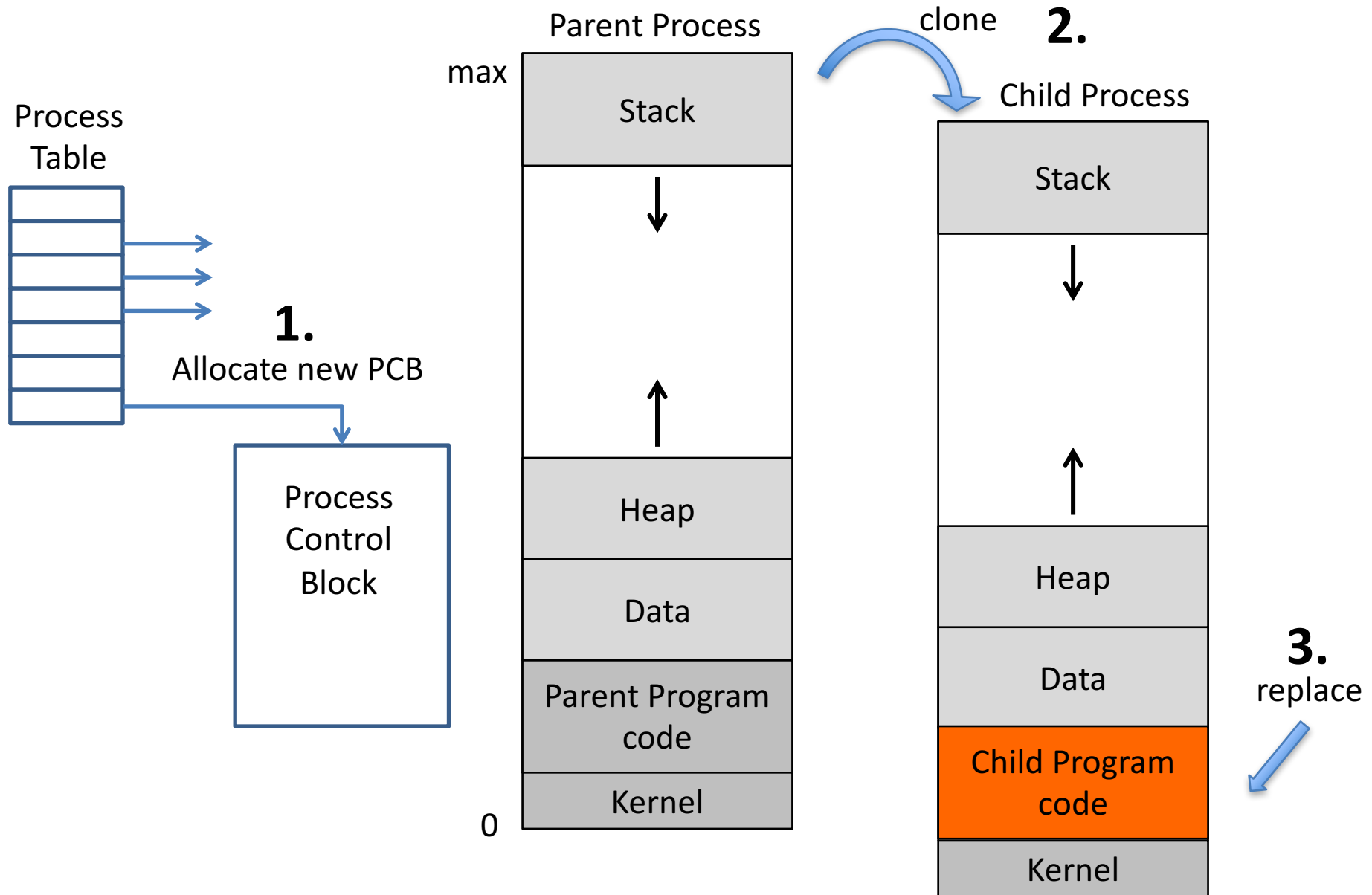
Unix Process Hierarchy



Process Creation

- Principle Events that lead to the creation of processes
 - System boot
 - An existing process spawns a child process
 - A process invokes a system call for creating a new process
 - e.g.: a server process (web server) may spawn a child process for each request handled
 - User request to create a new process
 - User types command in shell or selects widget in GUI to start a program, this creates a new process with the shell as the parent
 - A batch system takes on the next job in line (e.g. Jobs in a printer queue)

Process Creation in Unix



Process Creation under Unix

- A process creates new processes with the kernel system calls **fork()** and **exec()**
 - A new slot in the process table is allocated
 - A unique process ID is assigned to the child process
 - The process image of the parent process is copied, except the shared memory areas
 - Child process inherits the context of the parent
 - the same open files as parent
 - The same environment variables as parent
 - Child process is added to the Ready queue and will start execution

Programming Process Creation

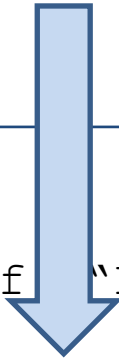
- System call **fork()**

```
int main( ... )
{
    int pid = fork() ;
    if ( pid == 0 )
    {
        // child process
        // program code for child
    }
    else if ( pid > 0 )
    {
        // parent process
        // program code for parent
    }

    return 0 ;
}
```

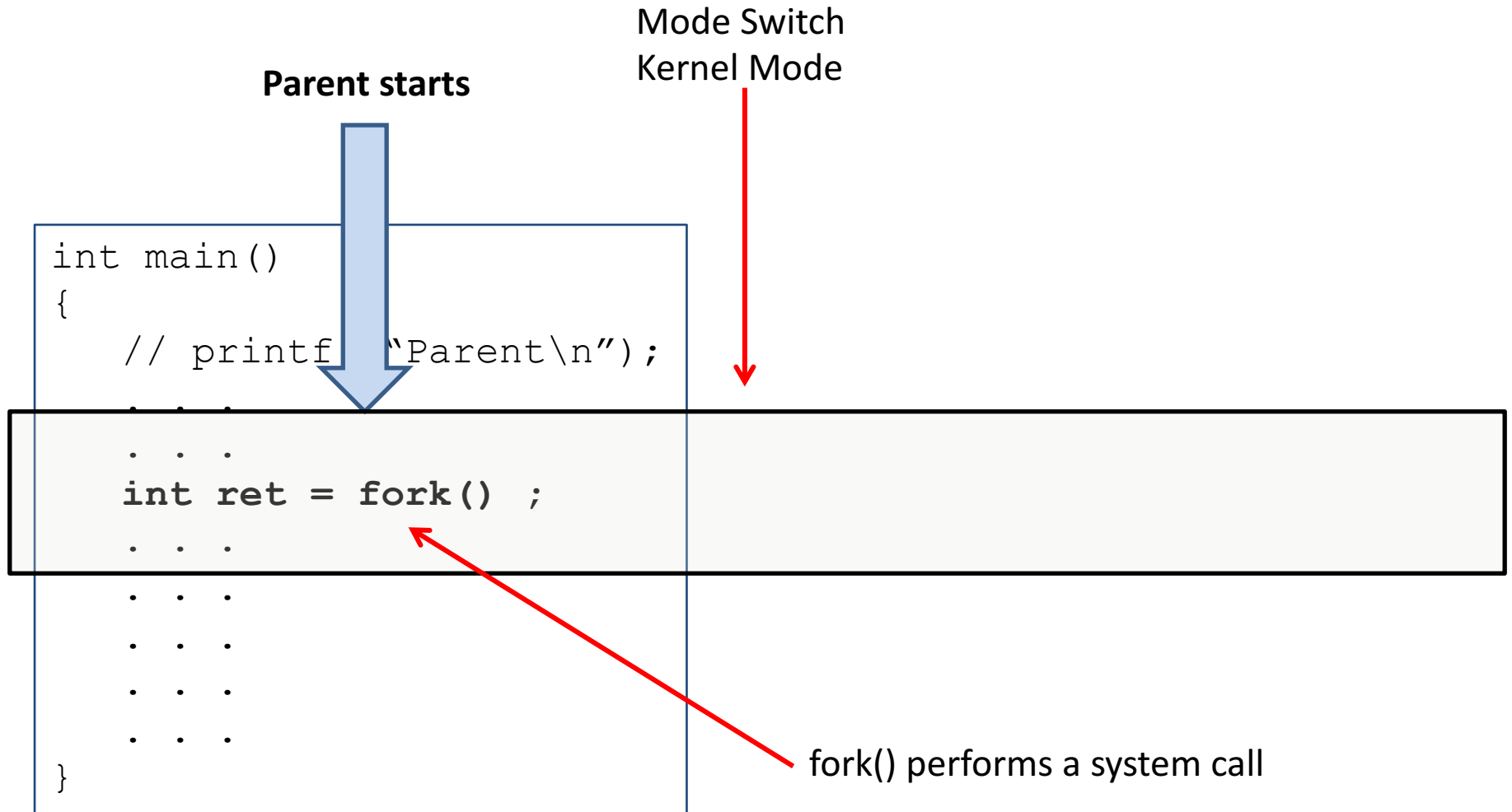
Process Creation under Unix

Parent starts

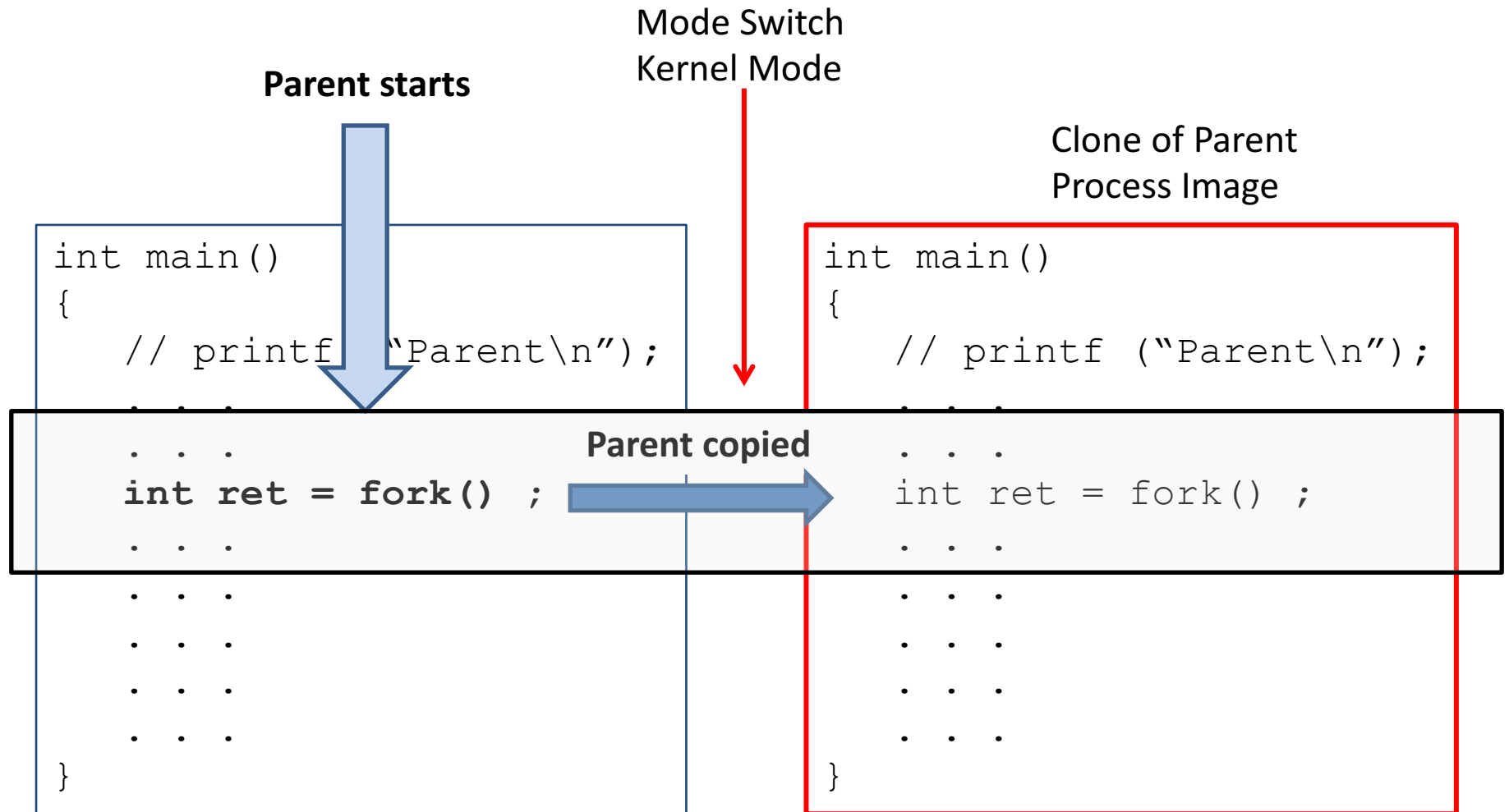


```
int main()  
{  
    // printf("Parent\n");  
    . . .  
    . . .  
    int ret = fork() ;  
    . . .  
    . . .  
    . . .  
    . . .  
    . . .  
}
```

Process Creation under Unix



Process Creation under Unix



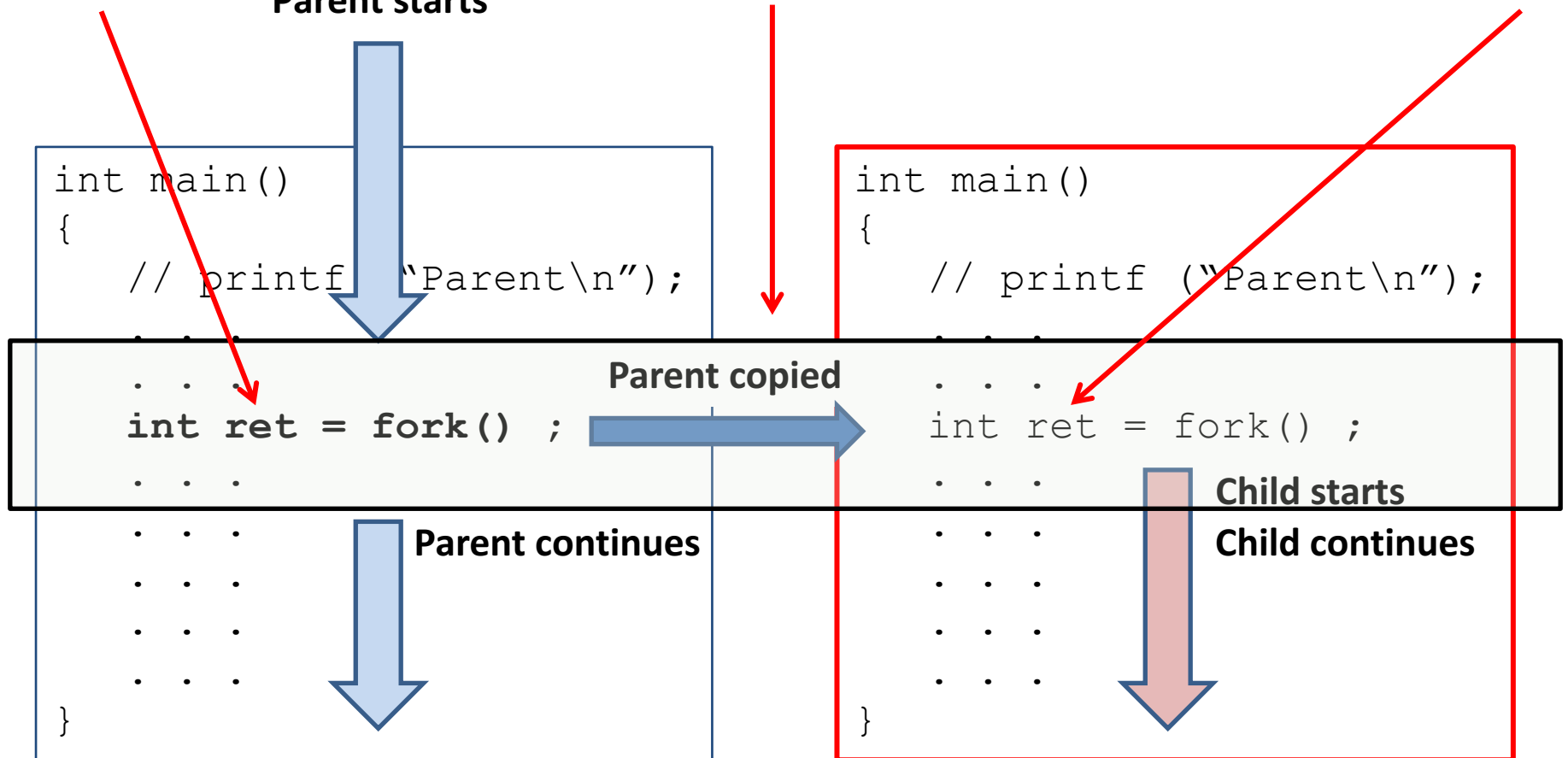
Process Creation under Unix

fork() return
value > 0

Parent starts

Mode Switch
Kernel Mode

fork() return
value == 0



Process Creation under Unix

- A paradoxical situation occurs:
 - When `fork()` is called by parent process, there is only one process
 - When `fork()` is finished and returns a return value, there are two processes
 - Child process is clone of parent process, with that, also the call of `fork()` has been cloned
 - Parent process continues execution at the return from calling `fork()`
 - Child process begins executing at the same point in the code as parent – at the return from calling `fork()`

Process Creation under Unix

- Distinction between parent and child process
 - System call `fork()` has different return value in parent and child processes
- Return value of `fork()`
 - Parent process: it returns the process ID of the child process
 - Child process: `fork()` return 0

Programming Process Creation

- Our program code has to serve the parent as well as the child process

```
int main( ... )
{
    int pid = fork() ;
    if ( pid == 0 )
    {
        // child process
        // program code for child
    }
    else if ( pid > 0 )
    {
        // parent process
        // program code for parent
    }

    return 0 ;
}
```


Programming Process Creation

Use of **exec()**

- Run a new program in child process image
 - Use of system call **exec()**

```
int main()
{
    // printf ("Parent\n");
    . . .
    . . .
    int ret = fork() ;
    if (ret > 0) {
        . . .
        . . .
        . . .
    }
    else if ( ret == 0 ) ...
}
```

Parent continues

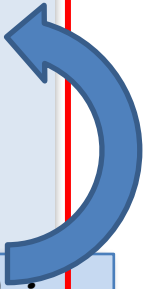


Parent copied



```
int main()
{
    // printf ("Parent\n");
    . . .
    . . .
    int ret = fork();
    if ( ret > 0 ) {
        . . .
    }
    else if ( ret == 0 )
    {
        exec ( "myprogram.exe" );
        . . .
        . . .
    }
}
```

New program



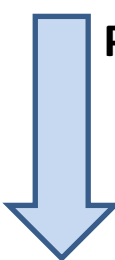
Programming Process Creation

Use of exec()

- System call `exec()` replaces content of cloned image with new program

```
int main()
{
    // printf ("Parent\n");
    . . .
    int ret = fork() ;
    . . .
    . . .
    . . .
    . . .
    . . .
}
```

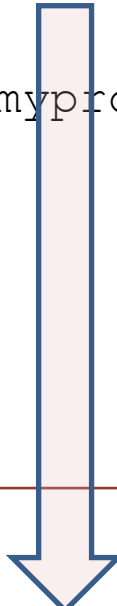
Parent continues



myprogram.exe

**New program starts
executing**

```
int main()
{
    // printf ("myprogram\n");
    . . .
    . . .
    . . .
    . . .
    . . .
}
```



System function exec()

- exec() comes in different flavours
 - execl()
 - execlp()
 - execlv()
 - execle()
 - execve()
 - execvp()
- Meaning of the name-annotations
 - “e”: an array of pointers to env variables is explicitly passed to the new process image
 - “l”: command-line arguments are passed individually
 - “p”: PATH env variable is used to find file name of program to be executed
 - “v”: command-line arguments are passed as an array of pointers

Example: Starting a Command in the Shell

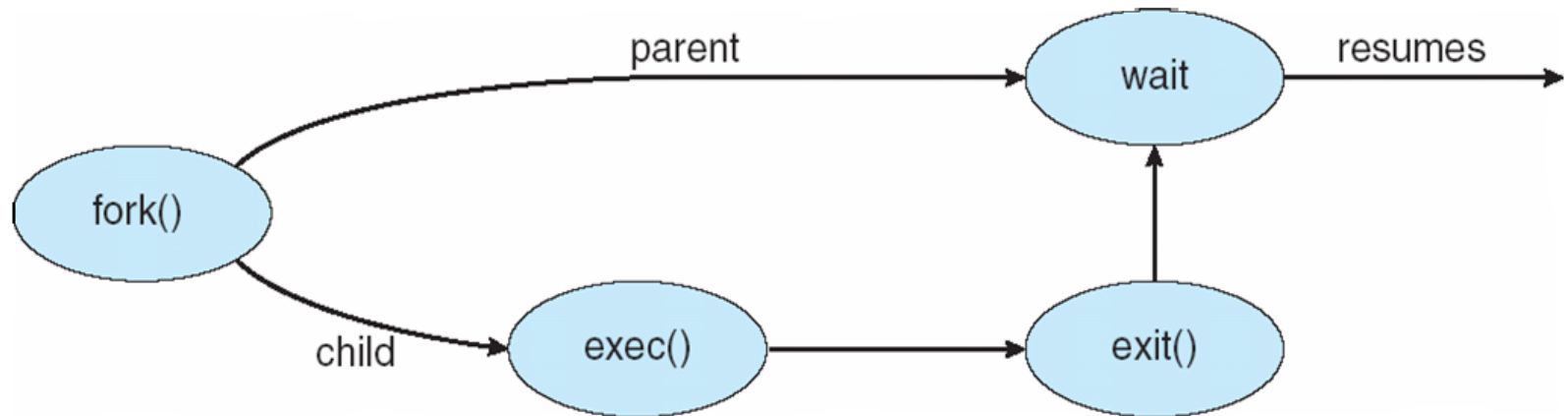
```
int main( ... )
{
    if ( (pid = fork()) == 0 ) /* child process */
    {
        printf ( "Child process id: %i\n", getpid() ) ;
        /* replace parent process image with executing a new program */

        err = execlp ( "/bin/ls", "ls", NULL ) ;

        fprintf ( stderr, "Child execution: %d\n", err ) ;
        exit ( err ) ;
    }
    else if ( pid > 0 ) /* parent process */
    {
        printf ( "Parent process id %d\n", getpid() ) ;
        /* parent will wait for child process to complete */
        child = waitpid ( pid, &status, 0 ) ;
    }

    return 0 ;
}
```


Process Synchronization with `wait()`



- Parent process may call system function `waitpid()` to wait for the exit of child process

Process Termination

- Normal termination
 - Program ends itself
- Abnormal termination
 - OS intervenes, user sends kill signal (CTRL-C)
 - Access to memory locations that are forbidden
 - Time out
 - I/O errors
 - Not enough memory, stack overflow
 - Parent process terminated
 - Etc.

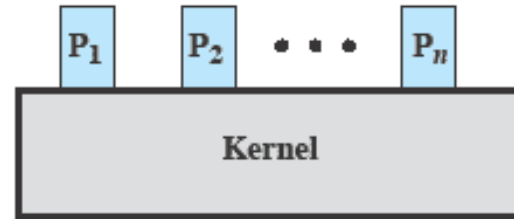
Process Termination

- Events
 - Normal exit (voluntary) / Error exit (voluntary)
 - Regular completion of a process, with or without error code
 - Process voluntarily executes the `exit(errno)` system call to indicate to the operating system that it has finished.
 - Fatal error (involuntary)
 - Uncatchable or uncaught
 - Service errors, such as: no memory left for allocation, I/O error, etc.
 - Total time limit exceeded
 - Arithmetic error, out-of-bounds memory access, etc.
 - Killed by another process via the kernel (involuntary)
 - The process receives a `SIGKILL` signal
 - In some systems, the parent takes down all its children with it

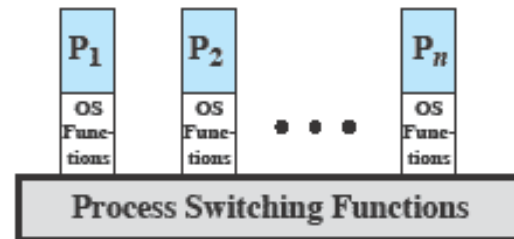
Execution of the Operating System

Execution of the Operating System

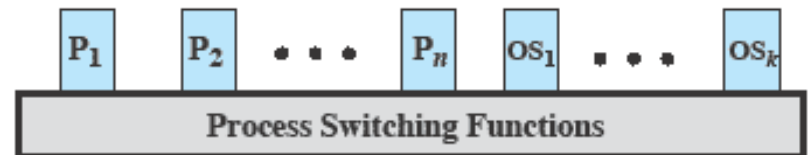
- Is the Operating System itself a process?
- Is it executing
 - Separately, only when processes are interrupted?
 - As part of the user process images?
 - As a set of processes (micro kernel)?



(a) Separate kernel



(b) OS functions execute within user processes



(c) OS functions execute as separate processes

Execution of the Operating System

- Is the Operating System itself running as a process? Or as a collection of processes?
- Various design options
 - Non-process kernel:
 - Traditional approach, many older operating systems
 - Kernel only executes when user processes interrupted
 - Execution of all OS functions in the context of a user process, mode switch
 - Process-based Operating systems, mode switch and context switch

Execution of the Operating System

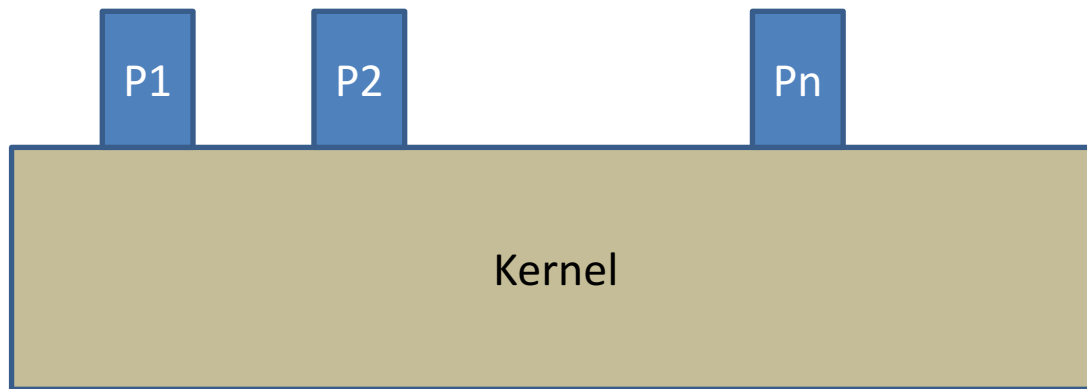
- There are different forms of kernels
 - Kernel operations separate from user operations
 - Only user programs are executing as processes
 - OS code executes in privileged mode, OS is a supervisor
 - All OS software is virtually executed within a user process, only Dispatcher outside user programs (process switching functionality)
 - OS is a collection of routines that are called by user programs
 - Mode switch occurs when user calls system routines
 - OS runs as a collection of system processes, dispatcher (process switching) is a small separate part

Non-process Kernel

- Many older operating systems
- Kernel acts as a monitor for user processes, only user processes regarded as processes
 - Kernel operations separate from user operations, is a supervisor
 - No concurrent execution of user processes and kernel
 - Processes are interrupted and control is switched back to kernel
 - Kernel executes only during these interruptions
- OS code executes in privileged mode , Operating system code situated within a reserved memory region, has its own system stack

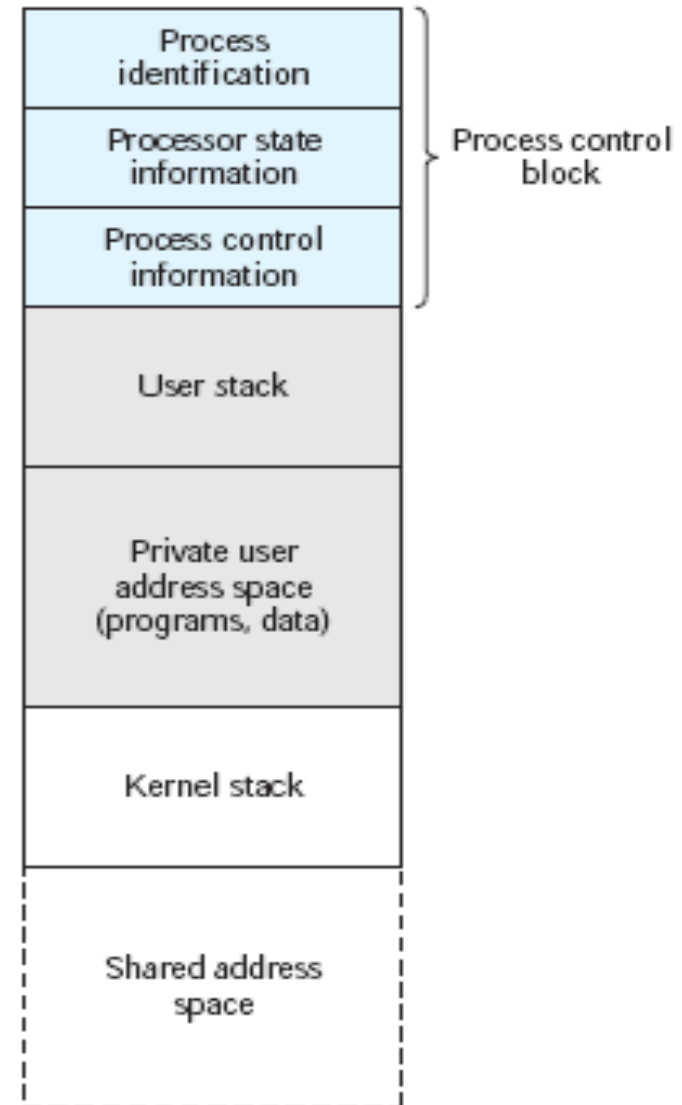
Non-process Kernel

- Kernel operations separate from user operations
 - Only user programs are executing as processes
 - OS code executes in privileged mode, OS is a supervisor



Execution within User Process

- User address space includes kernel functions
 - User address space “covers” kernel, can call system functions from within process
 - **Mode switch** necessary to execute these functions
 - **No** context switch, as we are still in the same process



Process-based Operating Systems

- Kernel functions run as separate processes, run in kernel mode
- Concurrency within kernel, kernel processes scheduled together with user processes
- Useful in multi-processor environments, as kernel functionality may be distributed to other CPU cores

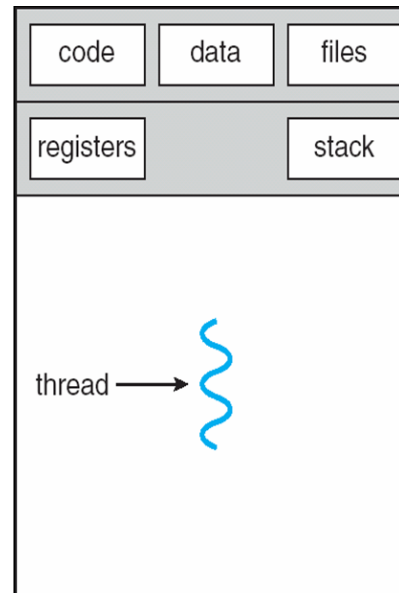
Threads

Multithreading

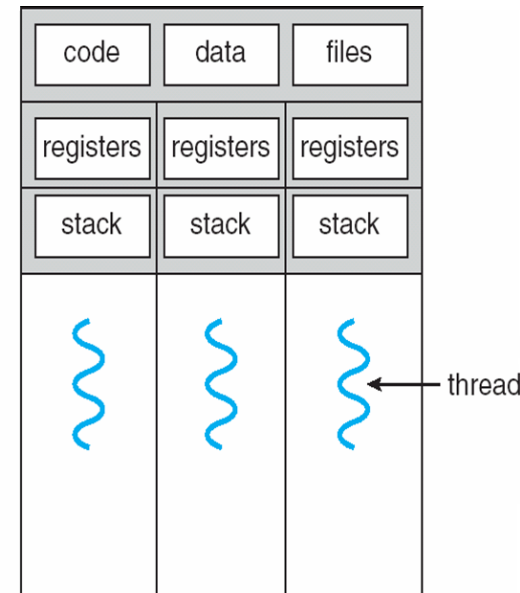
- Our process model so far: we defined a process as the **Unit of resource ownership** as well as the **Unit of dispatching**
- We want to separate these two concerns
 - **Resource ownership:**
 - Process remains unit of resource ownership
 - **Program Execution / Dispatching:**
 - A process can have multiple Threads of execution
 - Threads (lightweight processes) become the unit of dispatching
 - CPU may have multiple cores, true parallel execution of threads

Multithreading

- Multithreading is the ability of an operating system to support multiple threads of execution within a single process
- Processes have at least one thread of control
 - Is the CPU context, when process is dispatched for execution



single-threaded process



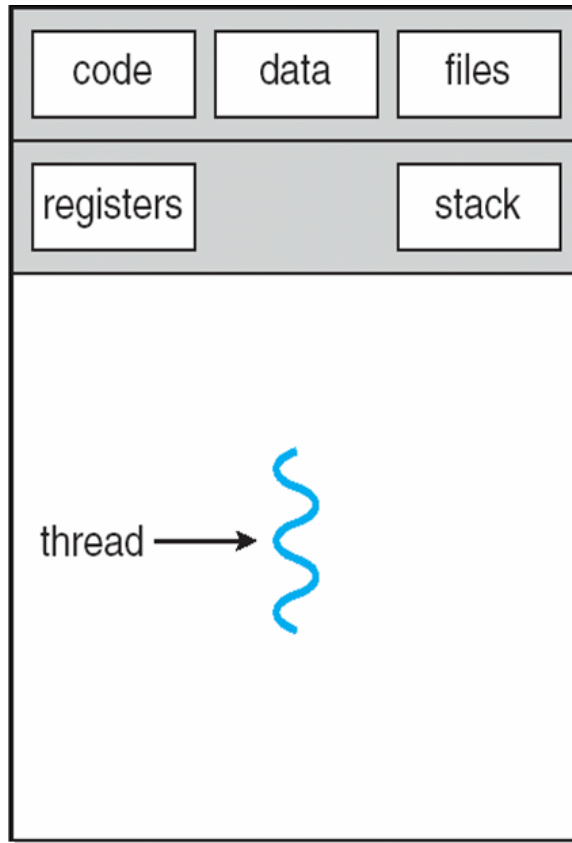
multithreaded process

- Multiple threads run in the same address space, share the same memory areas
 - The creation of a thread only creates a new thread control structure, not a separate process image

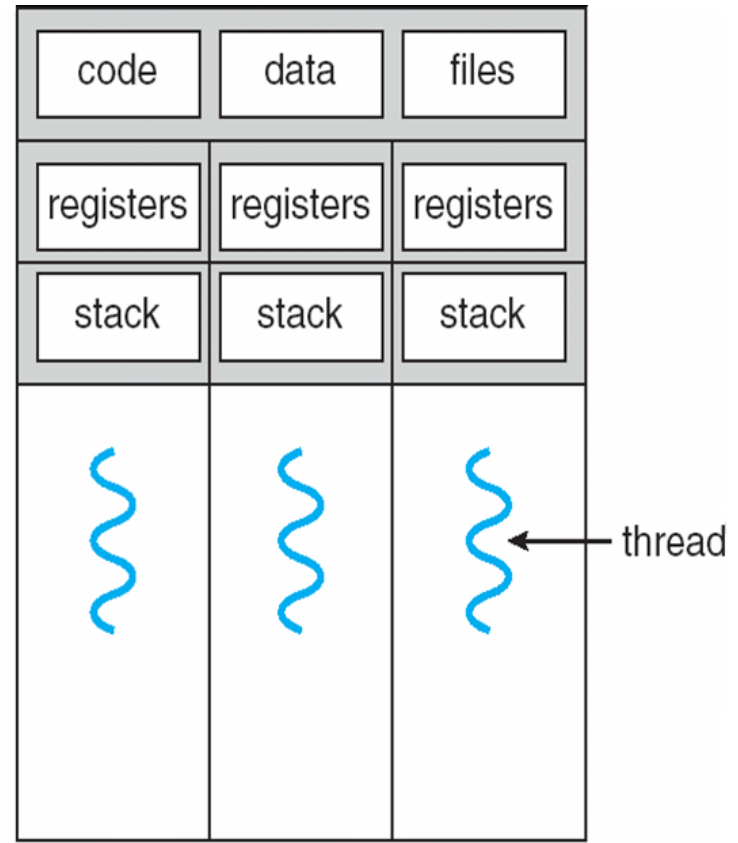
Multithreading

- Processes have at least one thread of control
 - Is the CPU context, when process is dispatched for execution
- Multithreading is the ability of an operating system to support multiple threads of execution within a single process
- Multiple threads run in the same address space, share the same memory areas
 - The creation of a thread only creates a new thread control structure, not a separate process image

Multithreaded Process Model



single-threaded process



multithreaded process

Process – Resource Owner

- Unit of **resource ownership** and **protection**
 - *Resource ownership*:
 - Process image, virtual address space
 - Resources (I/O devices, I/O channels, files, main memory)
 - Protection
 - Processors, other processes
 - Operating system protects process to prevent unwanted interference between processes
 - memory, files, I/O resources

Threads – Units of Dispatch

- Thread is defined as the **unit of dispatching**:
 - Represent a single thread of execution within a process
 - Operating system can manage multiple threads of execution within a process
 - The thread is provided with its own register context and stack space
 - Threads are also called “lightweight processes”

Threads – Units of Dispatch

- Thread is defined as the **unit of dispatching**:
 - Represent a single thread of execution within a process
- Threads are also called “lightweight processes”
 - Operating system may be able to manage multiple threads of execution directly (e.g. Linux tasks)
- A thread is provided with its own register context and stack space
- Multiple threads run in the same address space, share the same memory areas
 - The creation of a thread only creates a new thread control structure, not a separate process image

Threads

- All threads share the same address space
 - Share global variables
- All threads share the same open files, child processes, signals, etc.
- There is no protection between threads
 - As they share the same address space they may overwrite each others data
- As a process is owned by one user, all threads are owned by one user

Threads vs Processes: Advantages

- Advantages of Threads
 - Much faster to create a thread than a process
 - Spawning a new thread only involves allocating a new stack and a new thread control block
 - 10-times faster than process creation in Unix
 - Less time to terminate a thread
 - Much faster to switch between threads than to switch between processes
 - Threads share data easily
 - Thread communication very efficient, no need to call kernel routines, as all threads live in same process context

Threads vs Processes: Disadvantages

- Disadvantages
 - Processes are more flexible
 - They don't have to run on the same processor
 - No protection between threads
 - Share same memory, may interfere with each other
 - If threads are implemented as user threads instead of kernel threads
 - If one thread blocks, all threads in process block