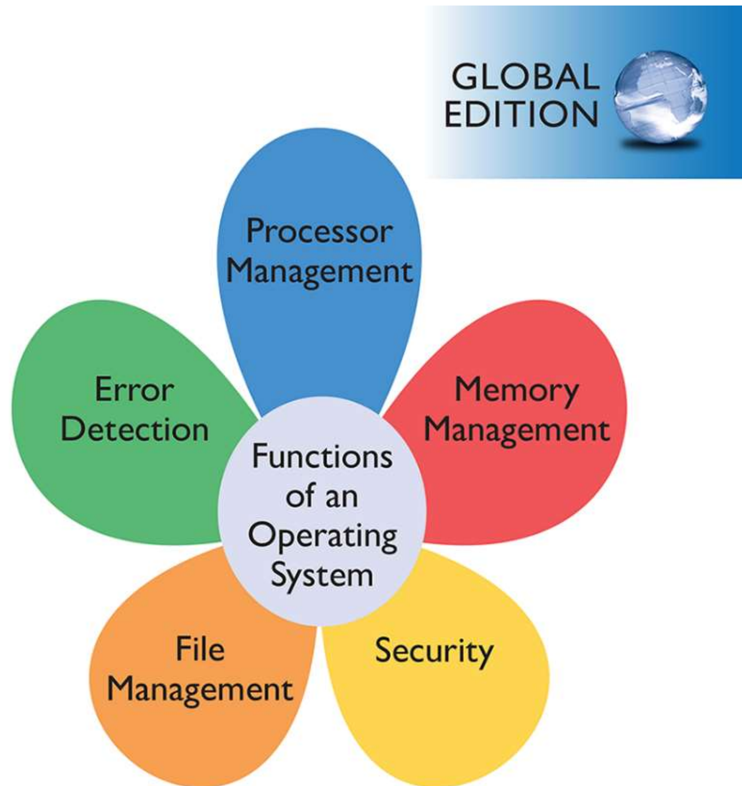
Modern Operating Systems

Fifth Edition, Global Edition



Lecture 2

Processes and Threads

Copyright © 2023 Pearson Education, Ltd. All Rights Reserved

OS Bootstrap (x86)

Basic I/O system (BIOS) initializes and executes the power on self test (POST). It queries buses and performs low level configuration of the peripherals (plug and play)

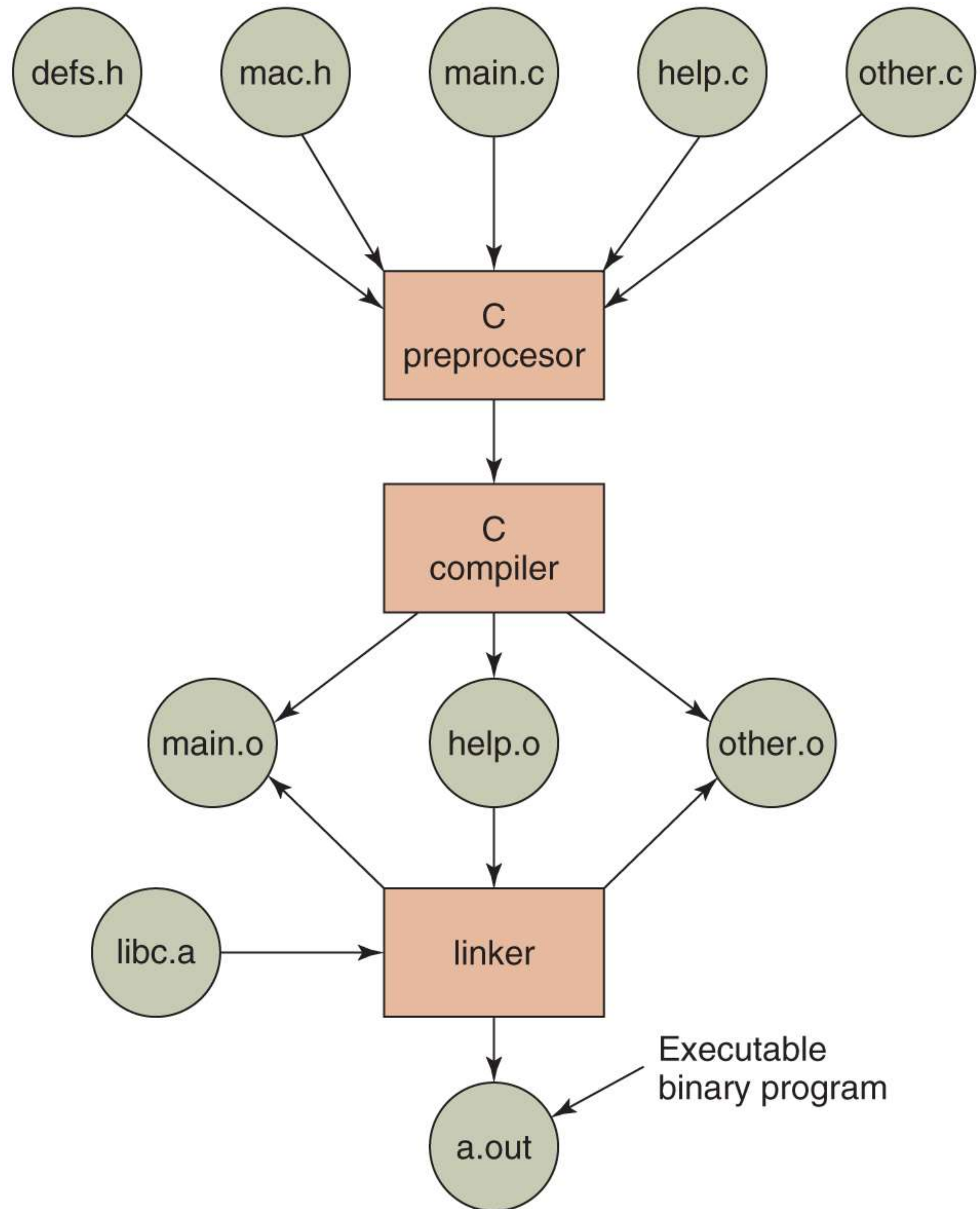
BIOS loads the boot sector of the designated I/O device The boot sector contains a small program which typically loads and executes a more sophisticated loader program.

In general, Bootloader program loads the OS and running its "init" process. Usually, a 2-stage bootloader is used for efficiency & security:

- first-stage (ROM/BIOS) is minimal, hardware-specific, and acts as a loader for the complex second-stage
- second stage (e.g., GRUB/U-Boot) provides filesystem support, network connectivity, and user interfaces

Compile

The process of compiling C and header files to make an executable.



The linker is usually invoked by the compiler transparently

Process & Process Control Block (PCB)

Process is a live program loaded in memory for execution. It is

- assigned resources (share mechanisms, acquire/release)
- run in user- or kernel-mode

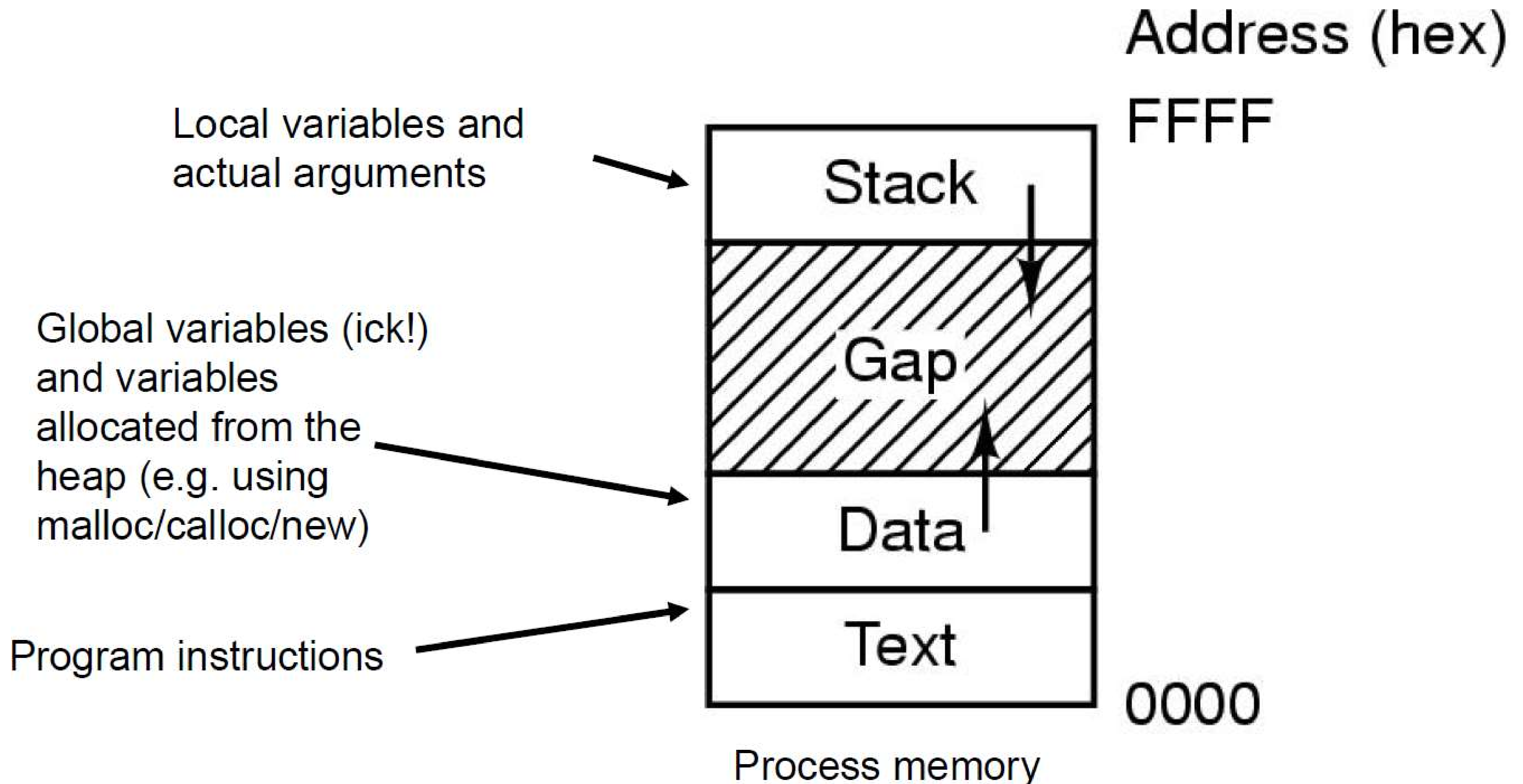
PCB includes code + variables + process ID (PID) + state + …

Many processes execute concurrently in a time-sharing OS

Note: Processes may have the same program, code, and data!

Process Control Block (Process Map)

Processes have three segments: text, data, and stack

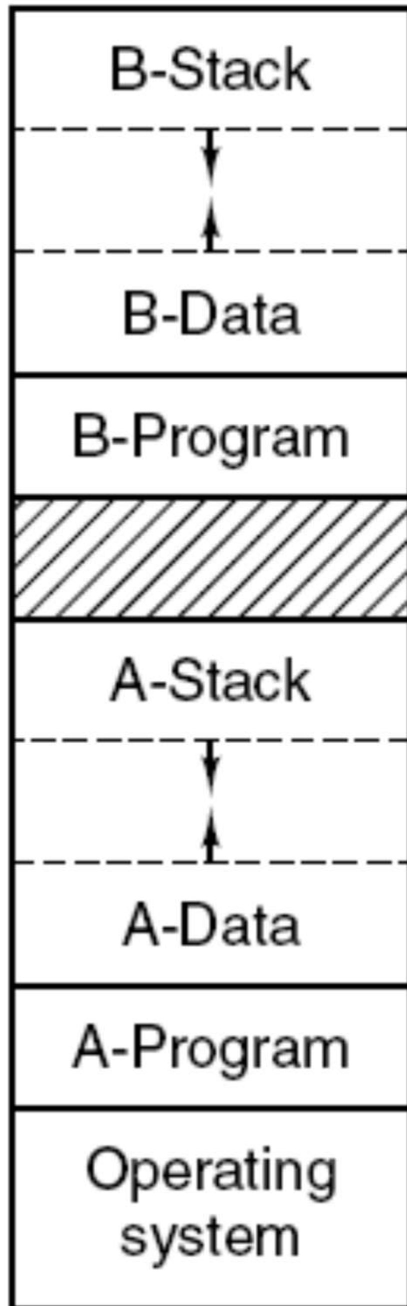


Process Control Block (PCB)

A typical PCB process-table entry is a kernel data structure representing processes. It contains may fields (state, id, etc):

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

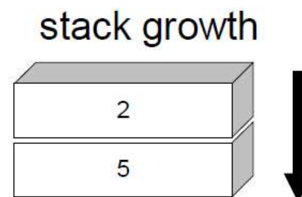
Process Maps



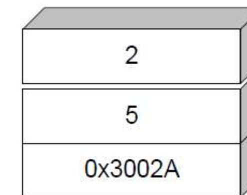
foobar(alpha, beta)

- let alpha = 5, beta = 2
- Push arguments
  (typically in reverse order)

stack growth



- Call function read
  push return address on stack
  execute jump subroutine instruction

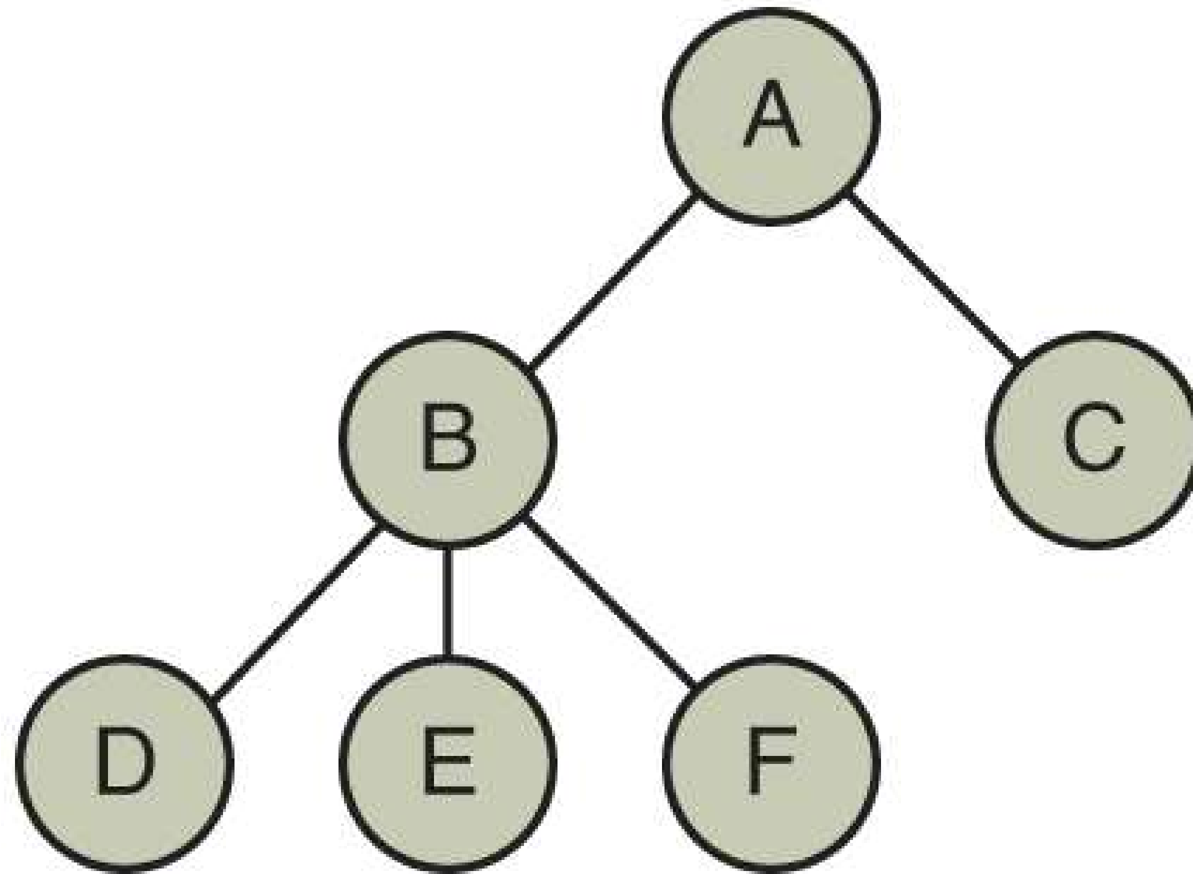


- Pop stack upon return

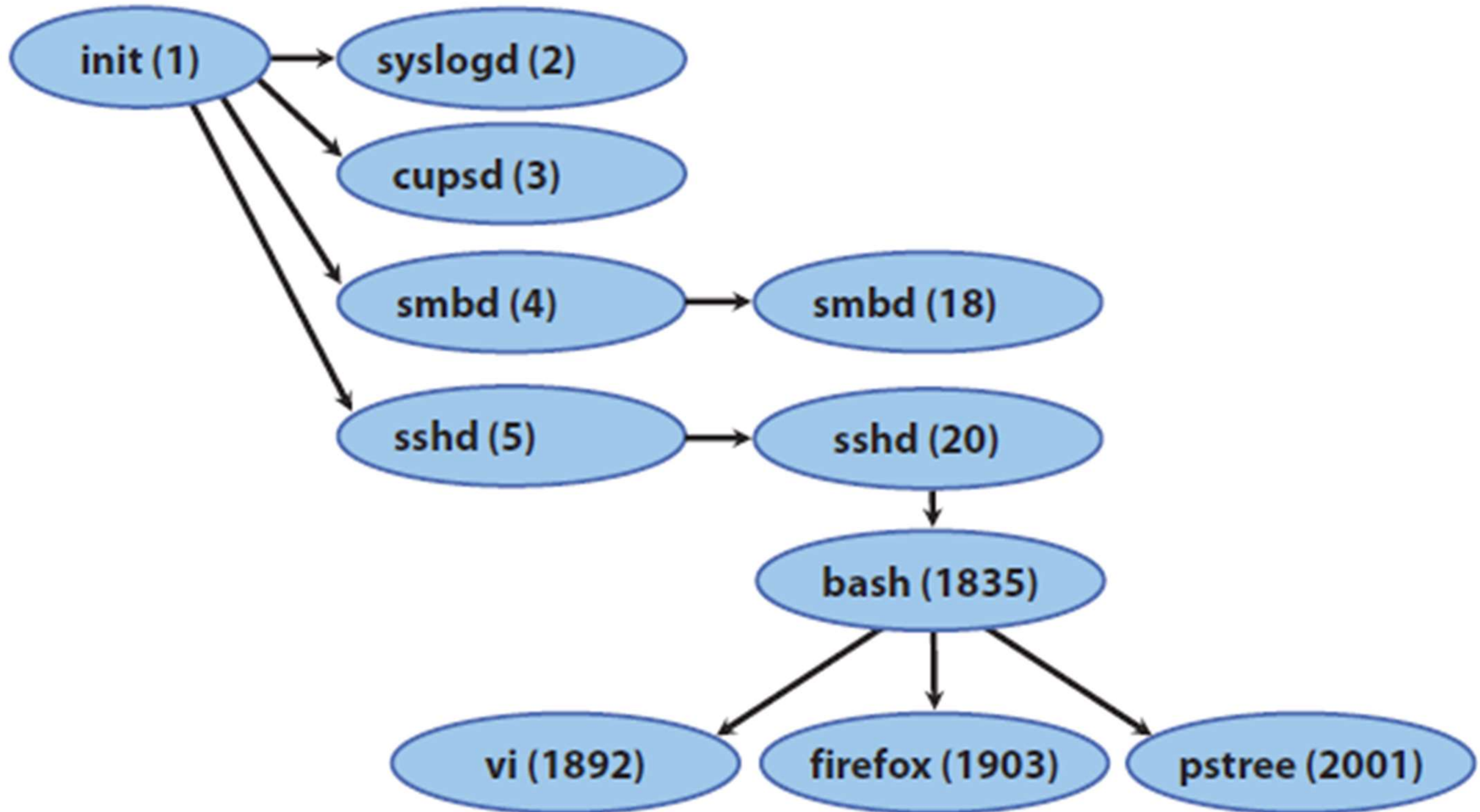
Process Tree

Process *A* created two child processes, *B* and *C*

Process *B* created three child processes, *D*, *E*, and *F*



Process Tree (Orphans, Zombies)



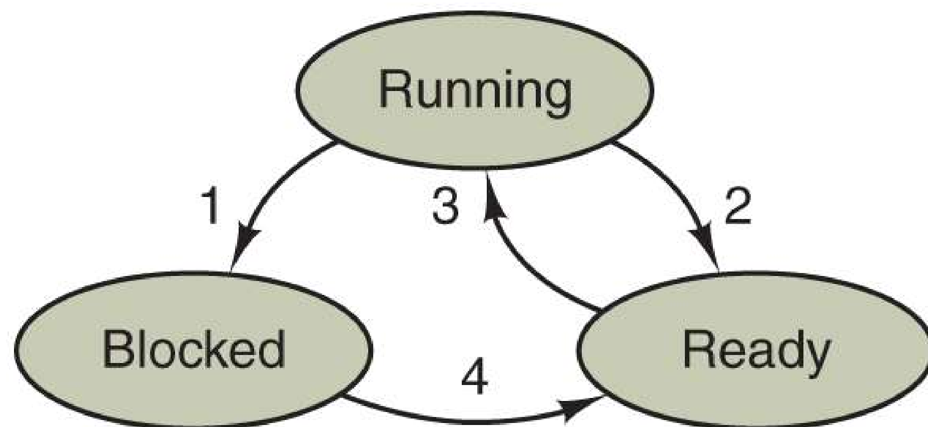
Lifecycle of a Process

A process is in running, blocked, or state

Transitions between these states are as shown

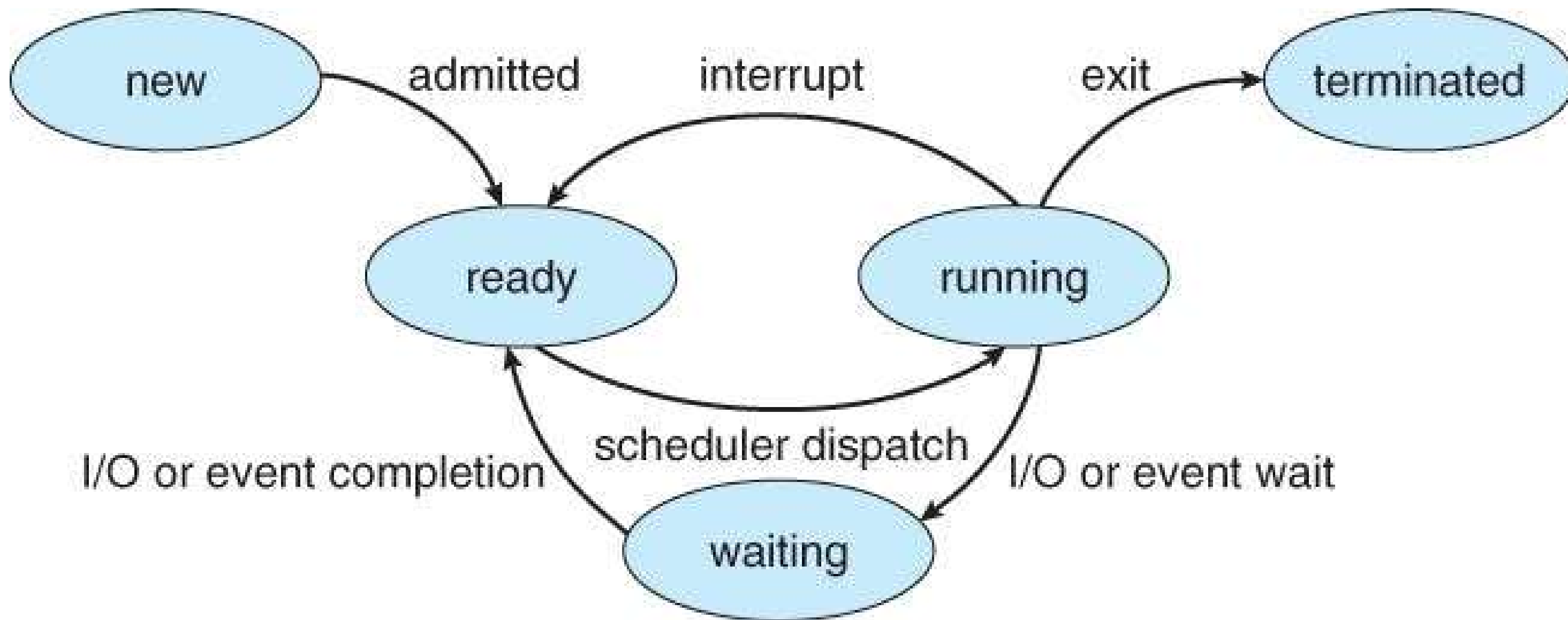
Note that initializing & terminating states are omitted

- System init by another process (init/systemd)
- Termination is voluntary (complete, error) or involuntary (exception, killed by another process)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Lifecycle of a Process



In Real Time Operating Systems: a higher-priority task instantly interrupts (preempts) a lower-priority task currently using the processor. The higher task is dispatched to run

Depending on the policy, this ensures that critical tasks meet their soft/hard deadlines, providing deterministic behavior

Real-Time Systems

Most real-time scheduling is in response to external events

- periodic – Occur regularly, e.g.,10 cycles/s
- aperiodic – We do not know when they will happen

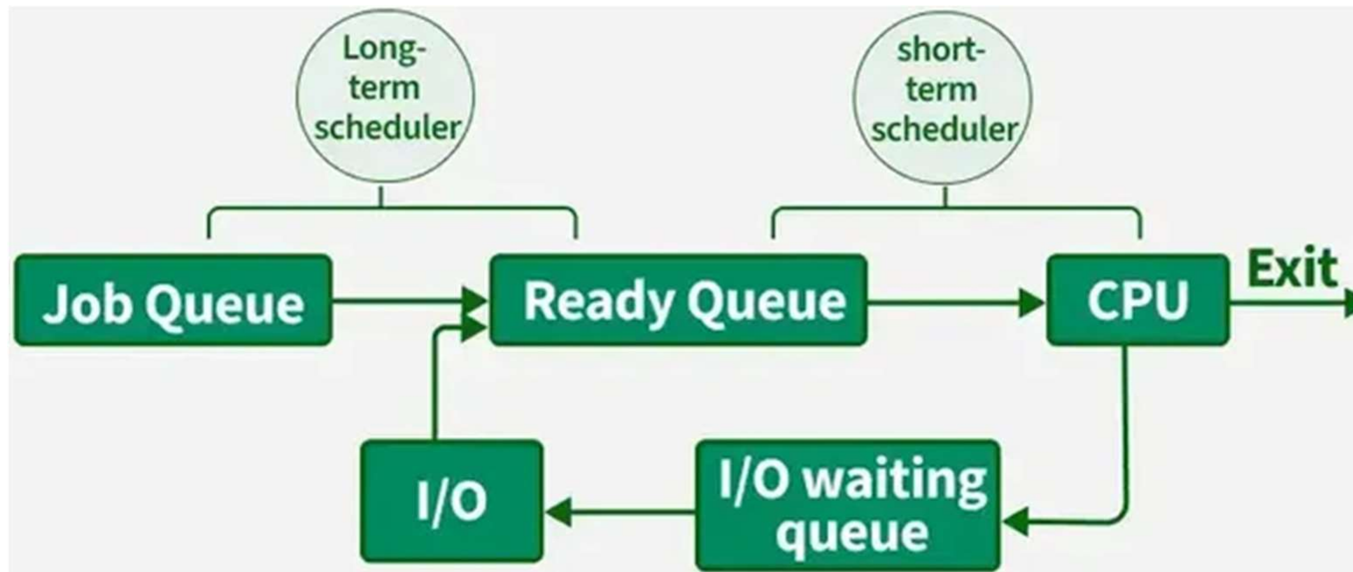
Real time systems must behave predictably (robust behavior for many environments)

Soft Real-Time: OS tries “really hard”

- Real-time processes assigned high priorities
- Frequently given separate queue

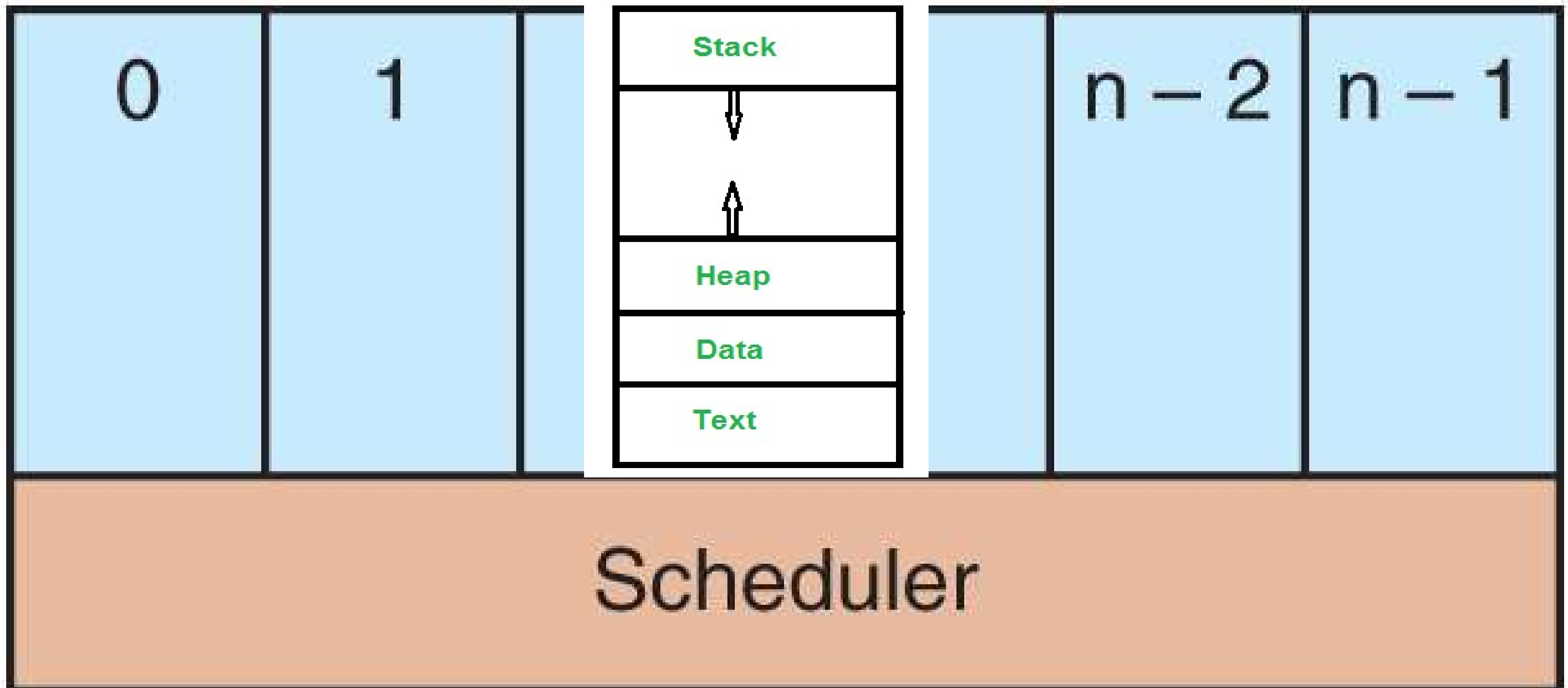
Hard Real-Time: processes request CPU time from OS. If granted, we guarantee that the deadlines are met

Lifecycle of a Process



Job Scheduling

The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.



Interrupt Request & Service Routine

What OS does when an interrupt occurs? The details may vary between operating systems.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Inter-Process Communication (IPC)

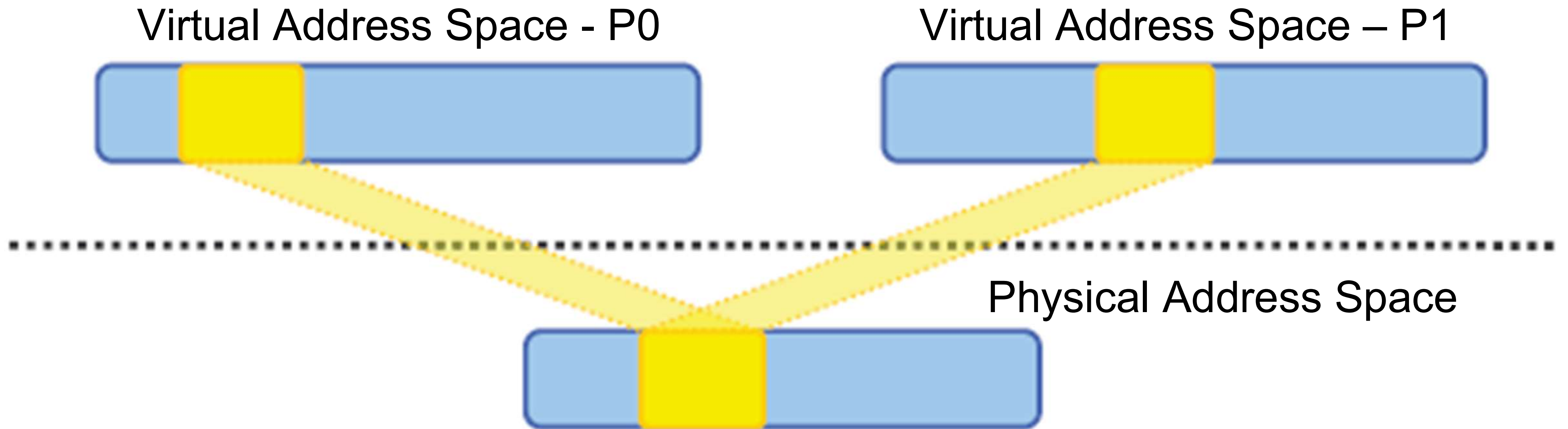
Cooperating processes

- Parallel execution, speedup
- Structured design, separation of privileges
- Concurrent access to shared data

Different IPC mechanisms

- Shared Memory (SystemV & POSIX Shared Memory)
- Message Queues (Microkernels, POSIX messages)
- Pipes (UNIX, Win32/64)
- POSIX Signals (UNIX)
- POSIX Sockets (UNIX)

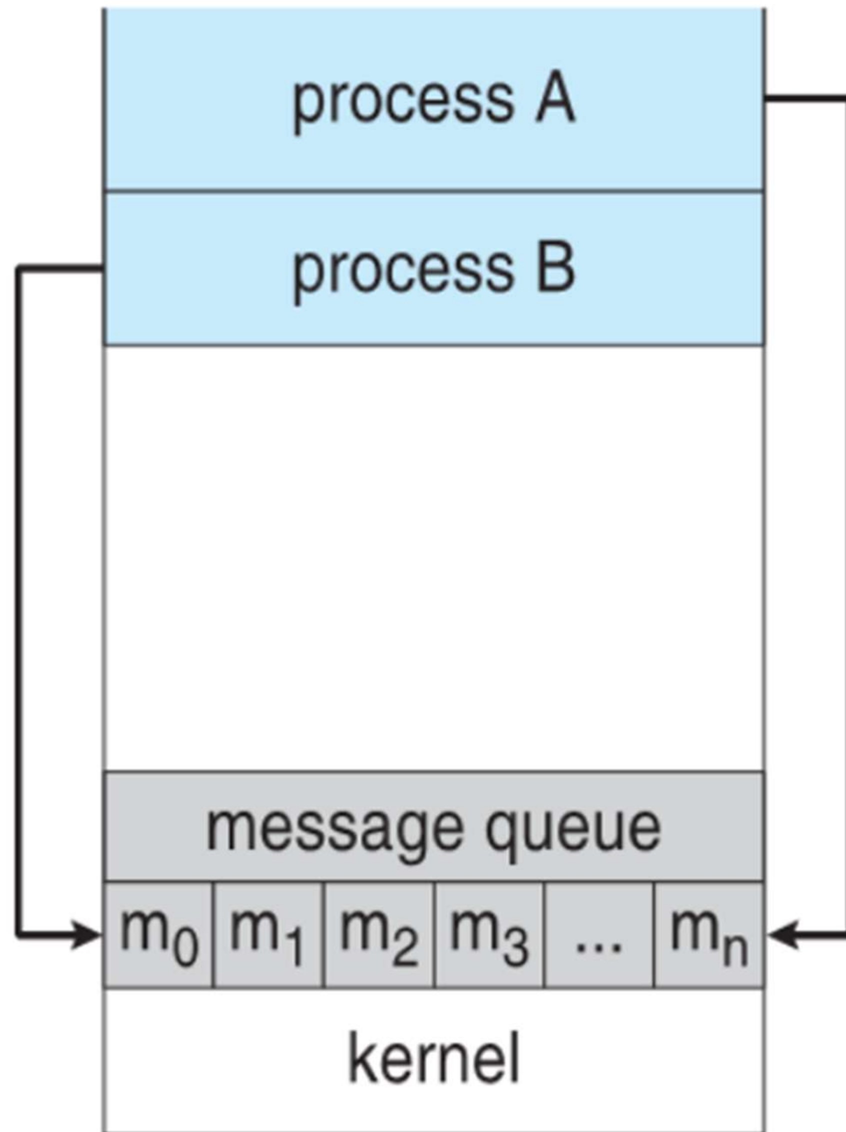
Shared Memory

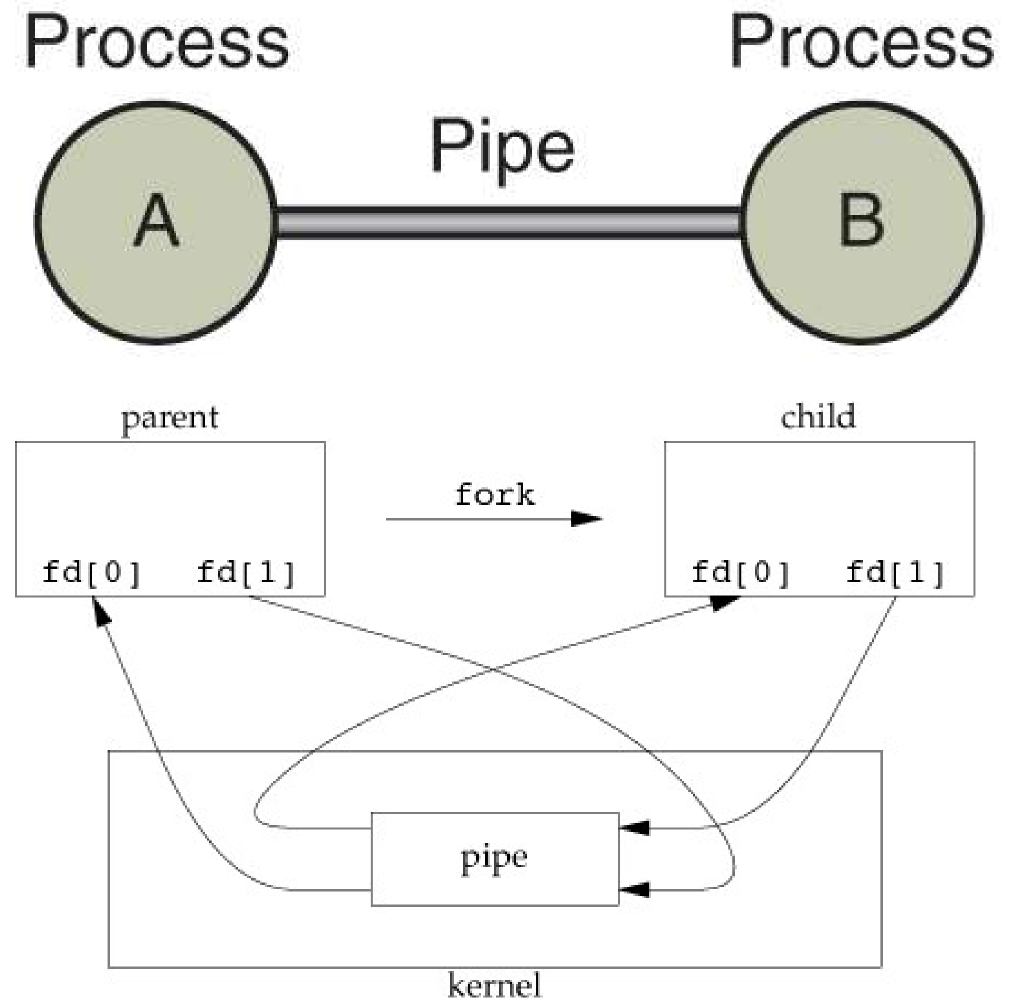


- Multiple processes access the same memory region directly (fastest IPC method)
- Communicate without kernel involvement after setup
- Processes must handle synchronization by themselves (e.g., mutexes, semaphores) to avoid *race conditions*
- Monitor changes via Polling

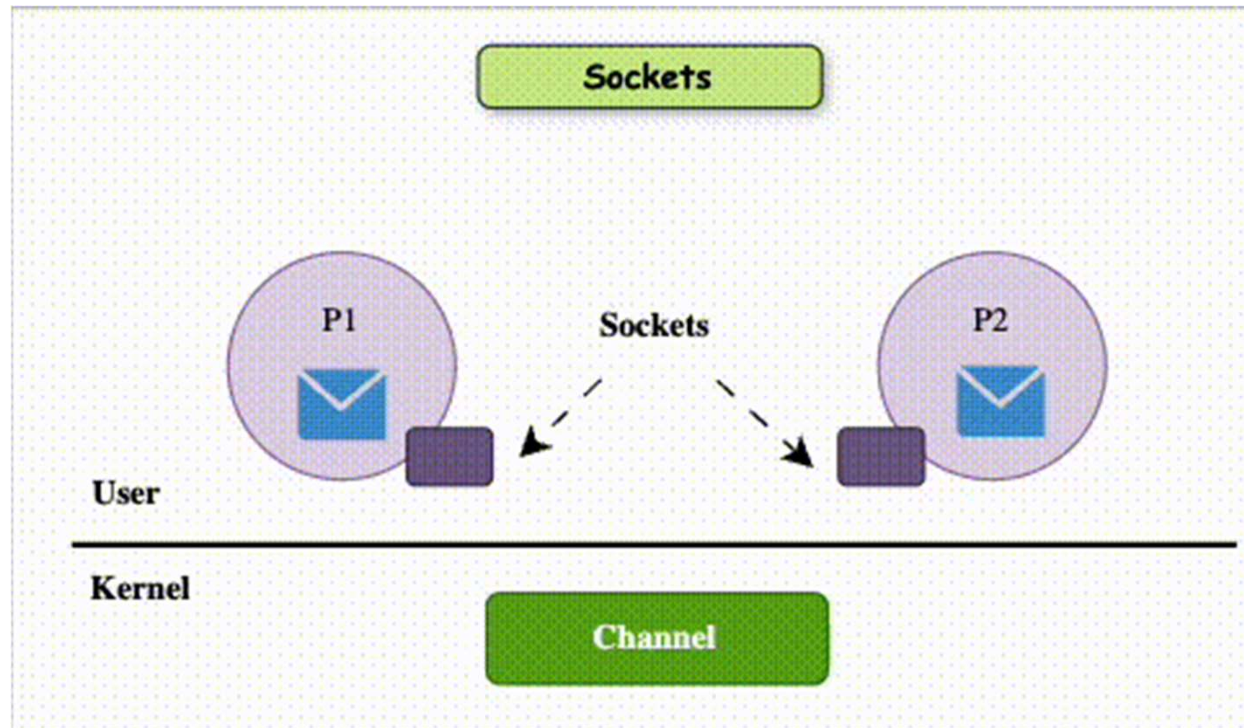
IPC via Message Queues & Pipes

Two processes connected by a Message Queue or Pipe.





Message Passing



Send/recv principles

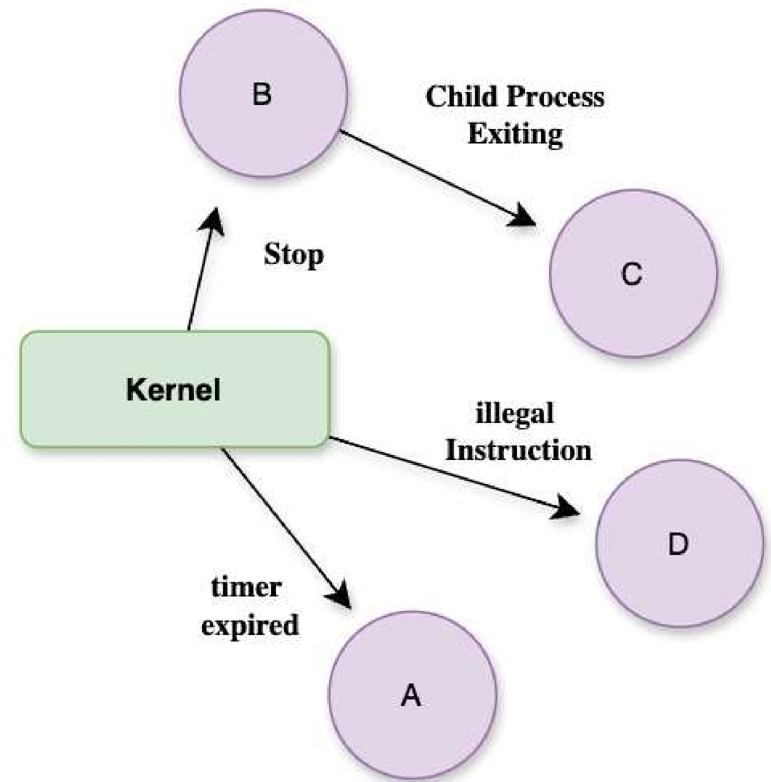
- Synchronous vs Asynchronous (Acknowledgments)
- Blocking vs Non-Blocking (Buffering)
- Synchronization

Unix Signals

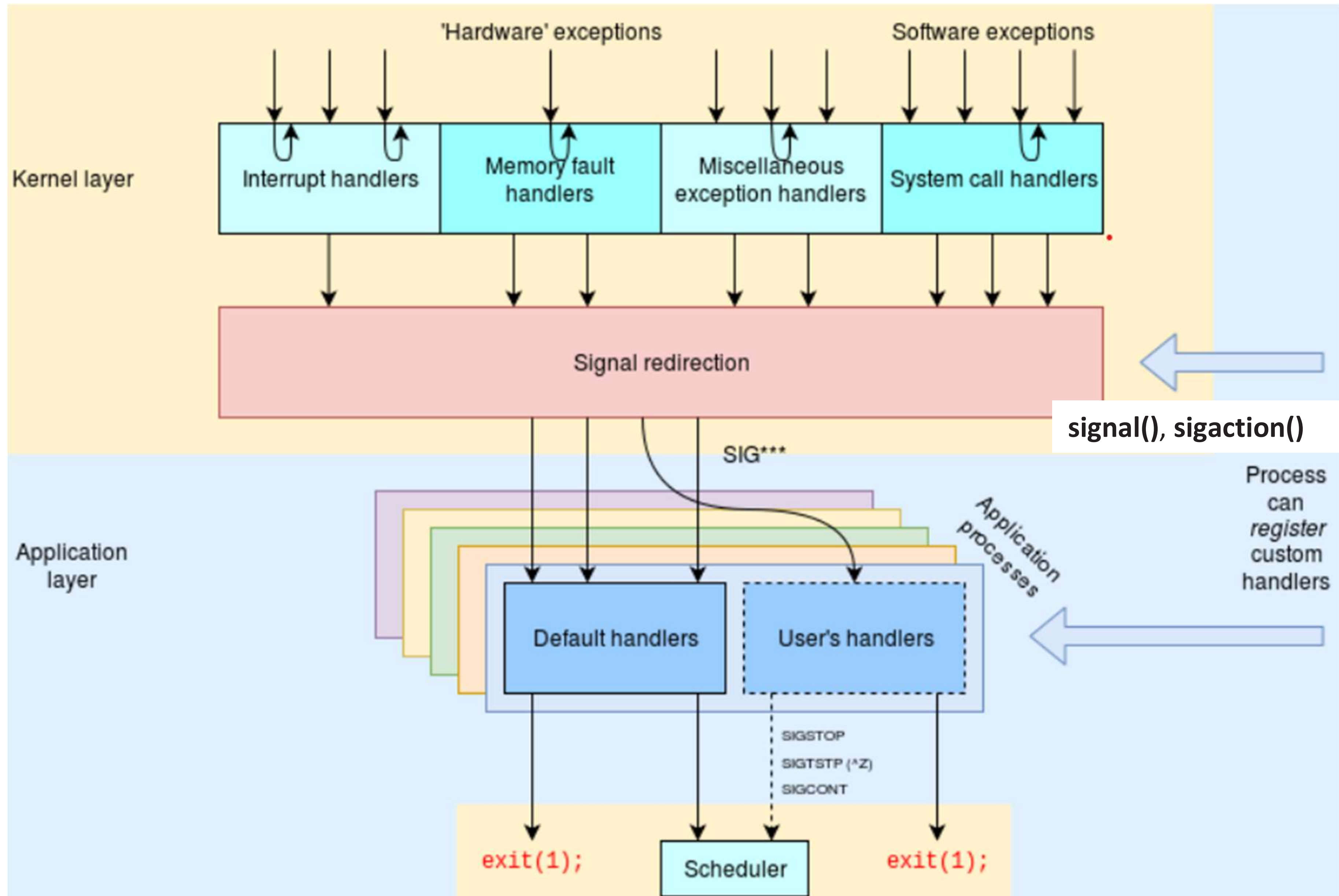
- Lightweight IPC used to notify processes on events, either from OS (e.g., SEGFAULT) or other processes (e.g., SIGKILL)
- Asynchronous notifications

Examples:

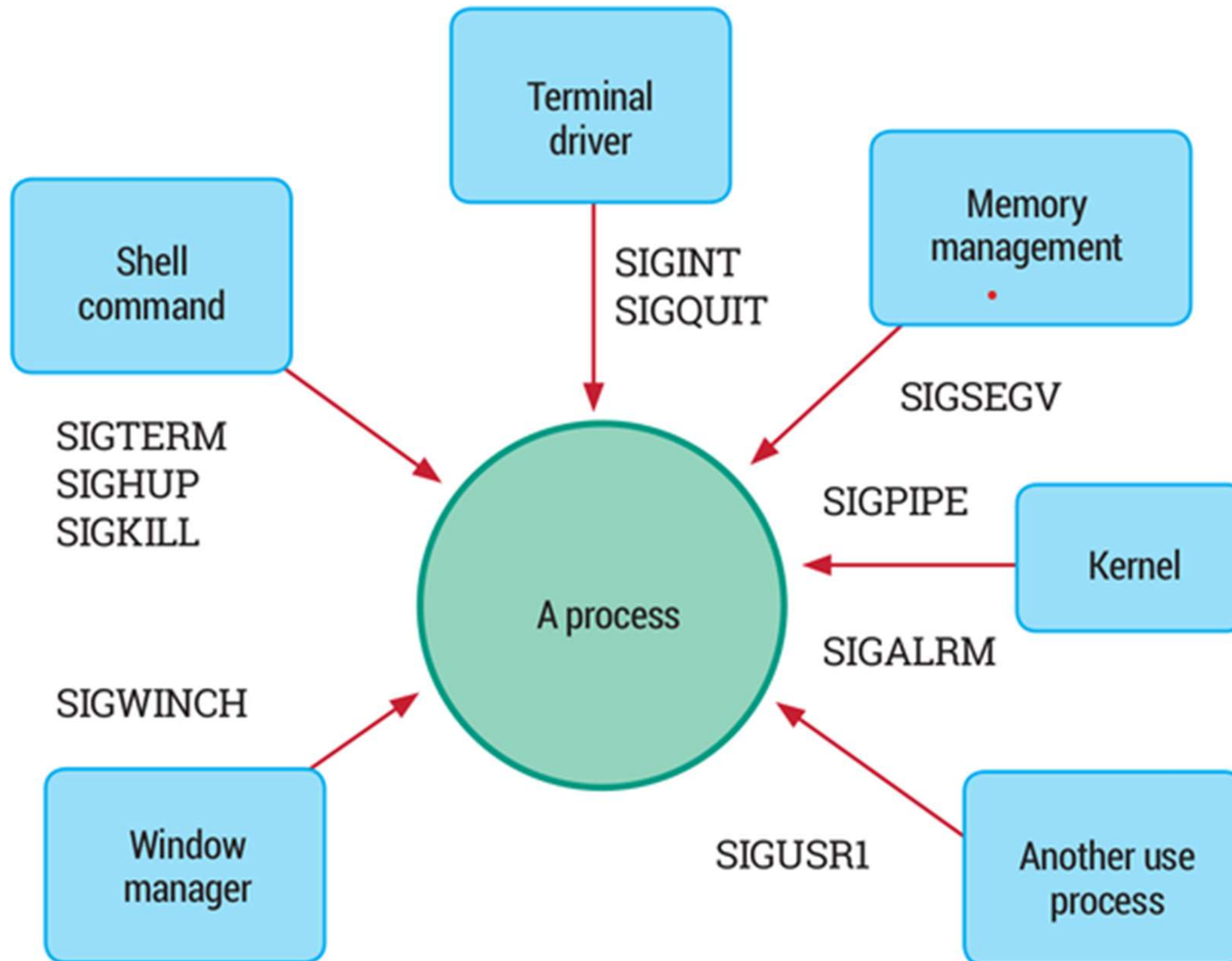
- SIGINT, SIGTSTP, SIGCONT, SIGSTOP, SIGFPE, SIGSEGV, SIGABRT, SIGQUIT, SIGWINCH, SIGTERM, SIGKILL



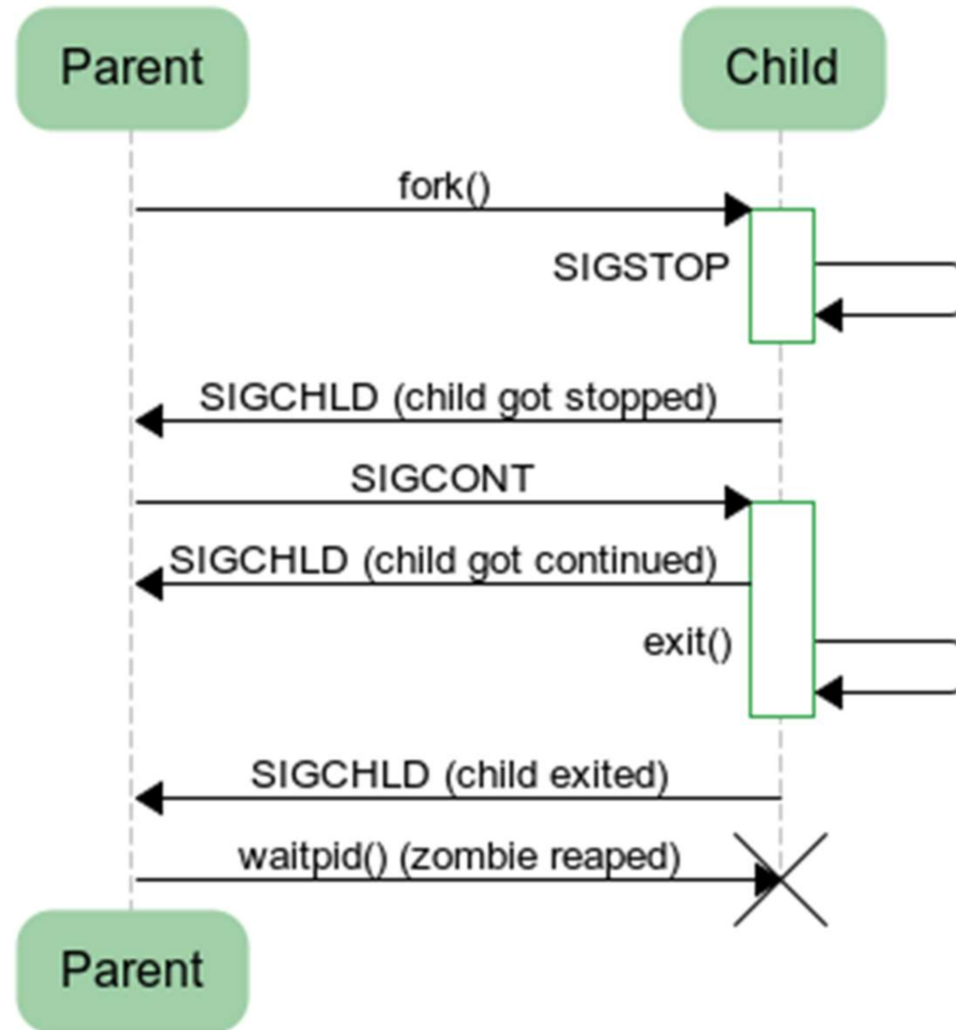
Unix Signals



Unix Signals



Unix Signals



- Parent-Child Signals

Unix Signals for Job Control

- Keyboard interrupt, Ctrl-C (SIGINT), SIGTERM
- Suspension from TTY, Ctrl-Z/bg/fg (SIGTSTP/SIGCONT)
- Termination & forceful termination (SIGKILL, SIGSTOP)
- SIGUSR1, SIGUSR2

Unix Signals (RT SIGNALS in 2001)

```
root@AJ7: /home/demon
File  Edit  View  Search  Terminal  Help
root@AJ7:/home/demon# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
root@AJ7:/home/demon#
```

Threads

Thread is an independent stream of instructions for execution. It includes

- CPU registers
- Stack
  - current procedure call
  - local variables

Threads are preferred over processes: they are lighter, faster, more responsive, and better suited for parallel and server-based applications

Multihtreading

- Multiple threads (aka lightweight processes)
- Threads within process:
  - are allocated the CPU, like processes
  - share resources: global variables, files, heap, etc
- Thread control block
  - Keeps registers, PC, PSW, state, …

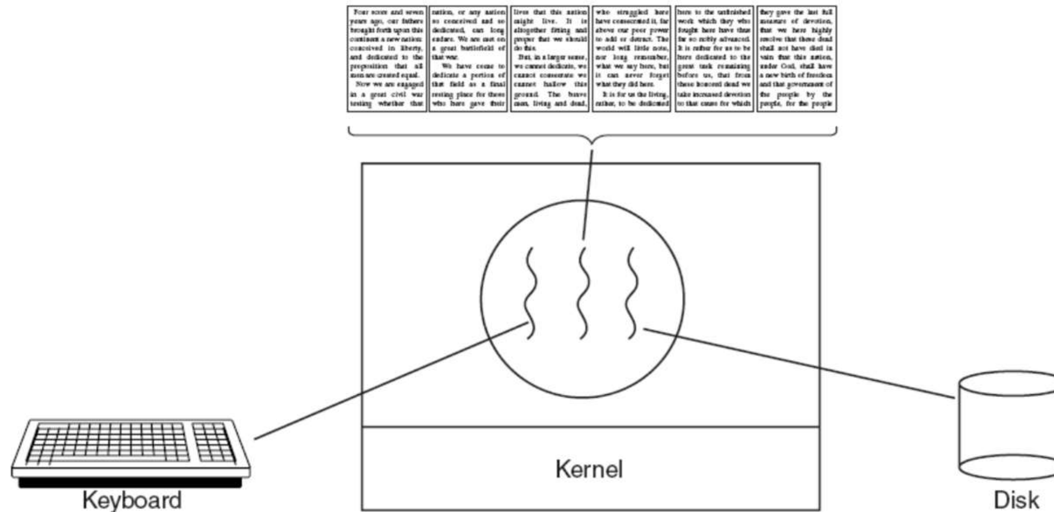
Thread Control Block

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Kernel: Process/Thread Control Block

```
struct task_struct {
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
void *stack; …
int prio, static_prio, normal_prio;
unsigned int rt_priority; …
struct mm_struct *mm, *active_mm; …
pid_t pid;  ...
/* open file information */
struct files_struct *files; ...
};
```

Thread Functions



- POSIX implementation (man pages/FAQ for details)
- headers: <pthread.h> <sched.h>
- pthread_create(…) – Create a thread
- pthread_yield(…) – Next thread runs
- pthread_exit(…) – Terminate thread
- pthread_join(…) – Wait for specific thread to exit
- pthread_attr_init(…) – Initialize options structure to be passed to pthread_create

POSIX pthread functions

Some of the Pthreads function calls

Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

POSIX pthread example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

POSIX Thread Example

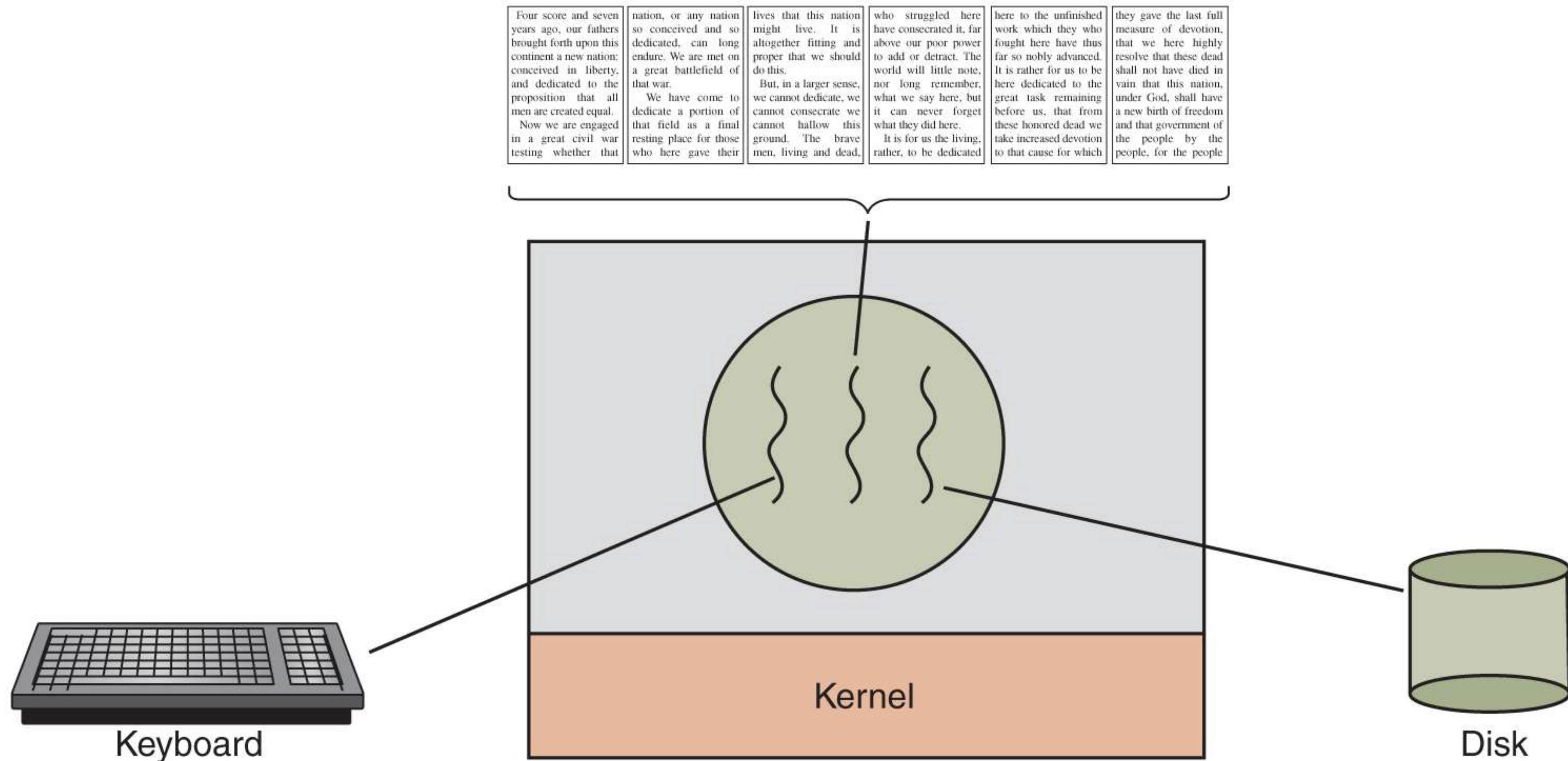
```
#include <pthread.h>

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Compilation: gcc thread.c –lpthread –lrt –o thread

Note: the order of libraries is important for static linking

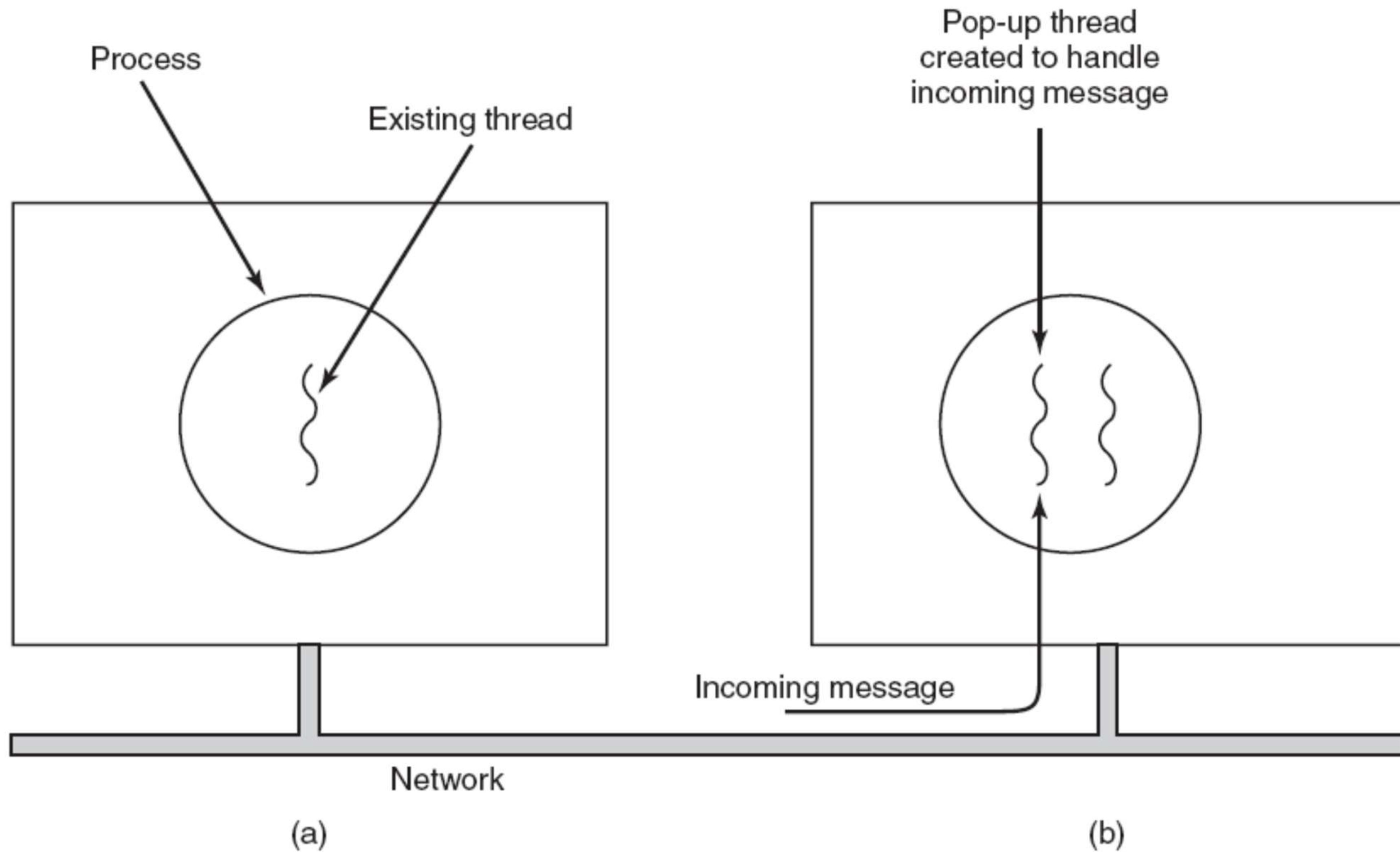
Thread Applications

A word processor with three threads.



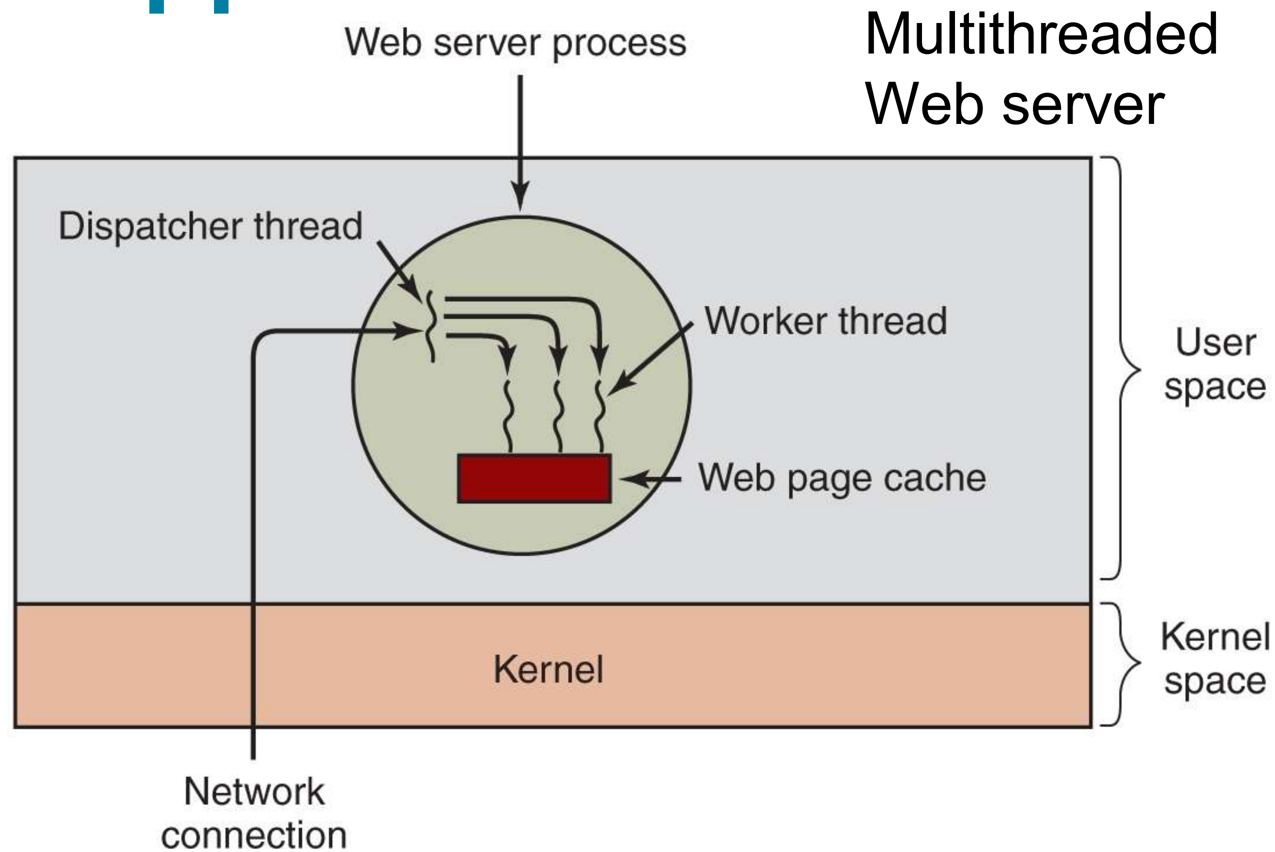
Spawing Threads

Dynamic creation of threads to handle events



Thread Applications

Multithreaded
Web server

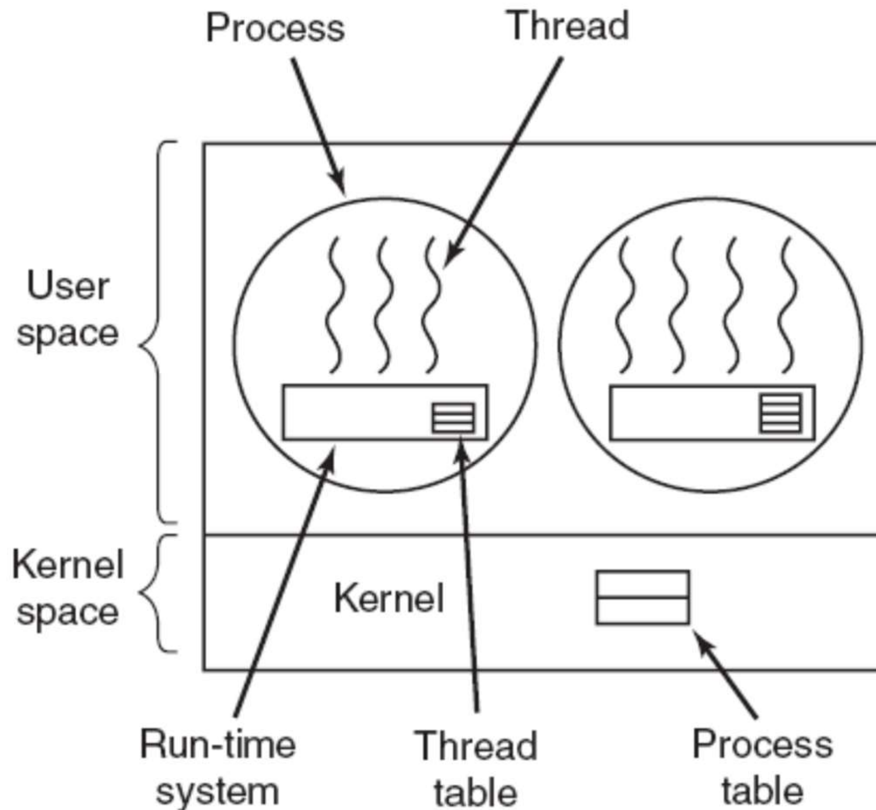


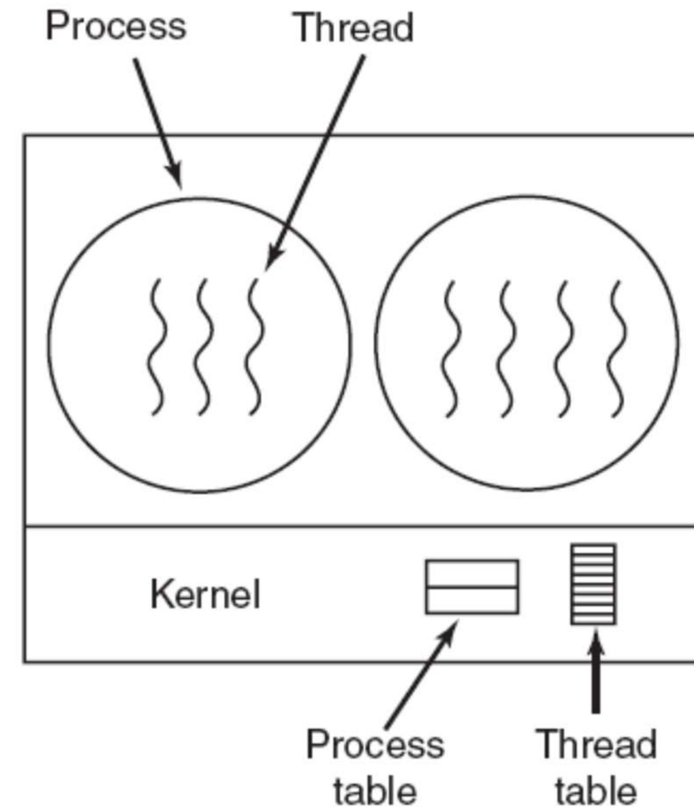

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(a) Dispatcher thread **(b) Worker thread.**

Threads: User- & Kernel-Level





User-level threads

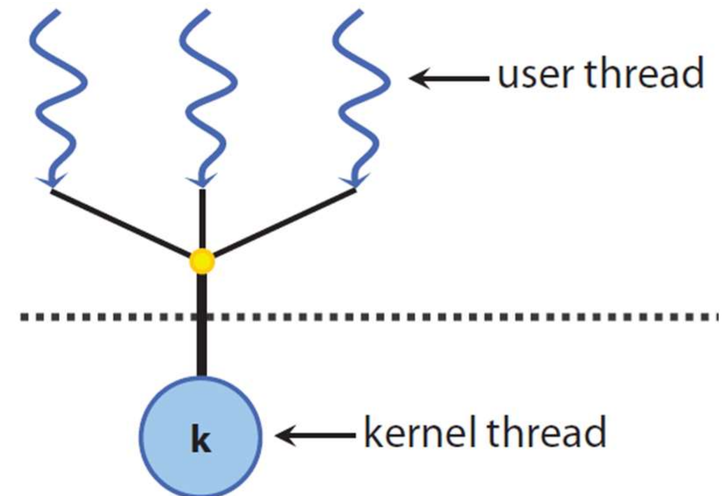
- implemented via a user library
- scheduling occurs in user code

Kernel-level threads

- part of OS implementation
- data structures are maintained in kernel code

Threads: User-Level Threads (ULT)

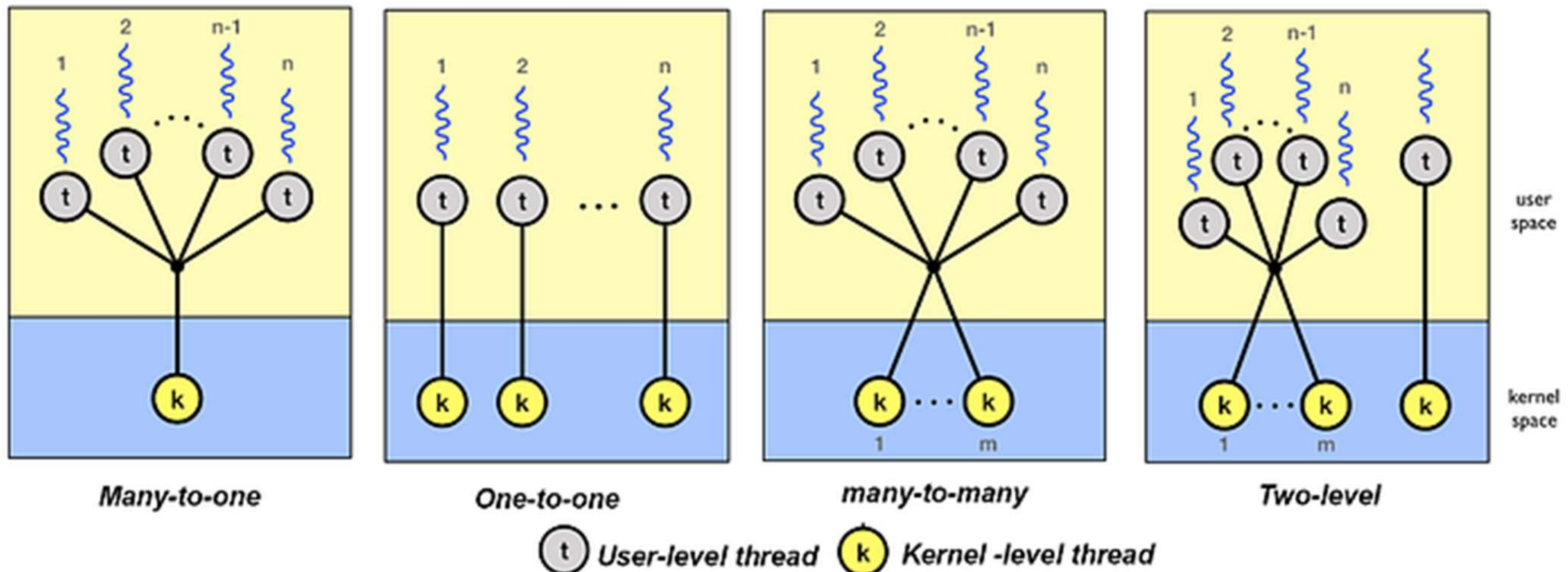
Multiplexing user-level threads
onto kernel-level threads



1-to-Many ULT case:

- Non-preemptive multitasking, threads voluntarily yield control
- Portability, low overhead for thread management
  - Very fast context switch: no kernel support, use POSIX calls makecontext() / swapcontext(), setjmp() / longjmp()
- Suitable for multithreaded servers
- Not for parallel computation (kernel schedules a single thread and system calls block all threads)
- Examples: nginx, node.js, go coroutines, Java threads

Threads: User-Level Threads (ULT)



- Managed user-level libraries or runtime environments
- Lightweight, fast creation and context switching, since no kernel intervention. However, if one ULT blocks, the entire process usually blocks. Examples include Argobots, massive threads (MP), and green threads

CPU Load

CPU utilization as a function of the number of processes in memory.