OPERATING SYSTEMS

#3

After A.S.Tanenbaum, Modern Operating Systems, 3rd edition

Uses content with permission from Assoc. Prof. Florin Fortis, PhD

PROCESSES

Basics

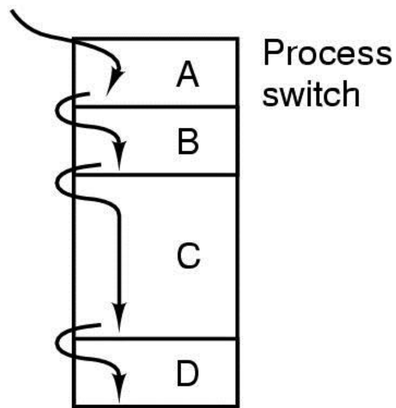
PROCESSES
BASICS

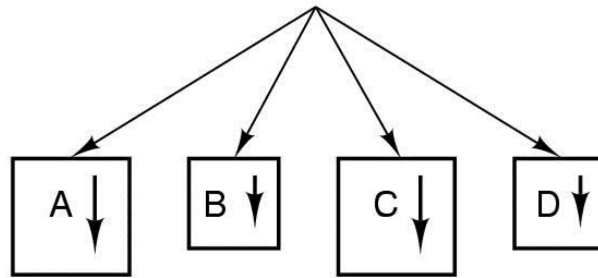
- A **process** is a program in execution, including all its associated values;
  - A system can hold at the same time several processes, including the operating system;
  - The OS offers the mechanisms needed for process switch operations, hiding process' sequential execution behind the illusion of parallelism;
- **Multiprogramming** is based on the process switch operation;
  - by using the process switch operation, the processor could be offered alternatively to every process;

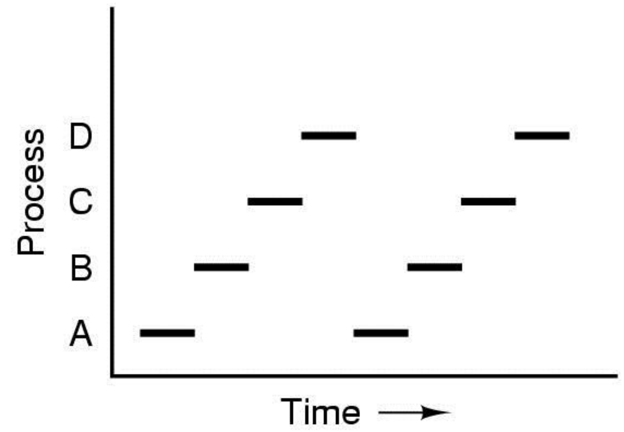
PROCESSES
BASICS

- In the case of process switch operation one should take into account the existing hardware limits:
  - There is only one set of registers are directly used for execution, even if the processor is able to offer more than one set of registers.
- The OS should use this issue in order to solve all the context switch operation (saving or restoring of contexts);
- The OS should not make any assumptions regarding the execution times of the processes.

PROCESSES BASICS







(a) Multiprogramming of four programs.
(b) Conceptual model of four independent, sequential processes.
(c) Only one program is active at once

PROCESSES
BASICS. PROCESS CREATION

- Events which cause process creation:
  - System initialization.
  - Execution of a process creation system call by a running process.
  - A user request to create a new process.
  - Initiation of a batch job.

PROCESSES
BASICS. PROCESS CREATION

- At system initialization time, the OS will create the processes needed in order to properly use the OS
  - E.g. processes that can support user interaction; processes designed for solving specified tasks.
- In modern OS new processes are able to create other processes, during their existence.
  - There are several reasons for this behavior:
    - offering of specialized services;
    - offering support for problems solving;
    - offering support for the execution of related processes, and others.

PROCESSES
BASICS. PROCESS CREATION

- For interactive systems, user requests can be used in order to create new processes.
  - User interaction is possible by using a command line interpreter; newly created processes are (usually) instances of system commands or user programs.
- For batch processing systems, process creation can occur while transmitting a job to the system, or as a result of a control card.
  - The OS should decide the creation of a new job only if the available resources are enough for this operation.

PROCESSES
BASICS. PROCESS CREATION

- The creation mechanism is as follows:
  - An existing process issues a system call in order to create a new process (e.g. the fork() system call). Some OS limit this possibility to certain categories of processes.
  - The OS could be announced, at the same time, about the need to load (and execute) another program in the newly created space (e.g. by using an exec…() system call).
  - Other operating systems, like Windows OS, could offer a unique (and complex) function, in order to solve all the issues at process creation time.

PROCESSES
BASICS. PROCESS TERMINATION

- Events which cause process termination:
  - Normal exit (voluntary).
  - Error exit (voluntary).
  - Fatal error (involuntary).
  - Killed by another process (involuntary).

PROCESSES
BASICS. PROCESS TERMINATION

- Most of the processes in a OS are characterized by a normal termination, once a specific system call is issued;
- Fatal errors represent special events during process execution (such as an invalid file name).
  - A process is not able to continue its execution due to a fatal error;
- Exceptions (like division by zero, illegal memory access, null pointer operations) offer another kind of reasons for process termination.
  - Exceptions can be intercepted and handling. For this kind of events, process termination can be avoided;
- Forced termination can occur only by a request of another process, authorized for this kind of requests.

PROCESSES
BASICS. HIERARCHIES OF PROCESSES

- Different OS offer some mechanisms in order to emphasize the relationships between processes.
  - UNIX systems are based on a parent-child relation for processes, too.
  - All the processes are descendants of a unique process, init, that was created during the system initialization process (traditionally, the PID – Process ID – value is 1).
  - For the Windows OS, the "*creator*" of a process has a special token that can be used in order to control the "*created*" process. However, in Windows' philosophy all processes are equal.

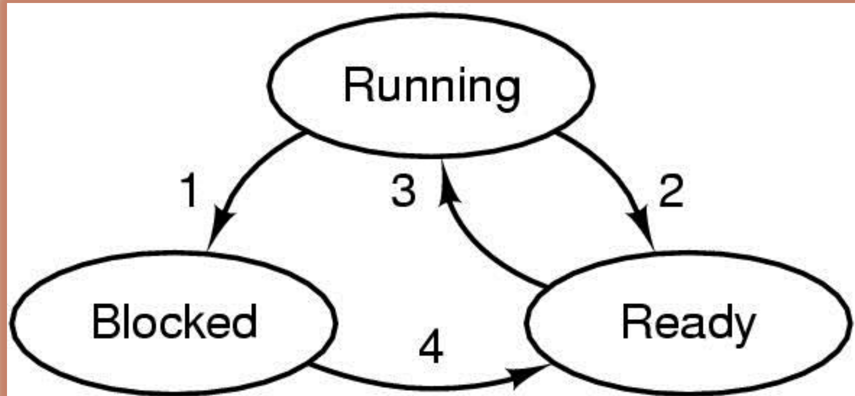
PROCESSES

Process' states

PROCESS' STATES
THE MODEL WITH 3 STATES

- The states of a process reflects the main moments during process execution. Early systems use a 3-states model:
  - "running"
  - "prepared"
  - "blocked".
- Processes are blocked while waiting for an external event to occur (such as, an I/O operation, the activity of a scheduler, and others)
- Each OS implement its own transition schema, according to system's policy for process management.

PROCESS' STATES
THE MODEL WITH 3 STATES



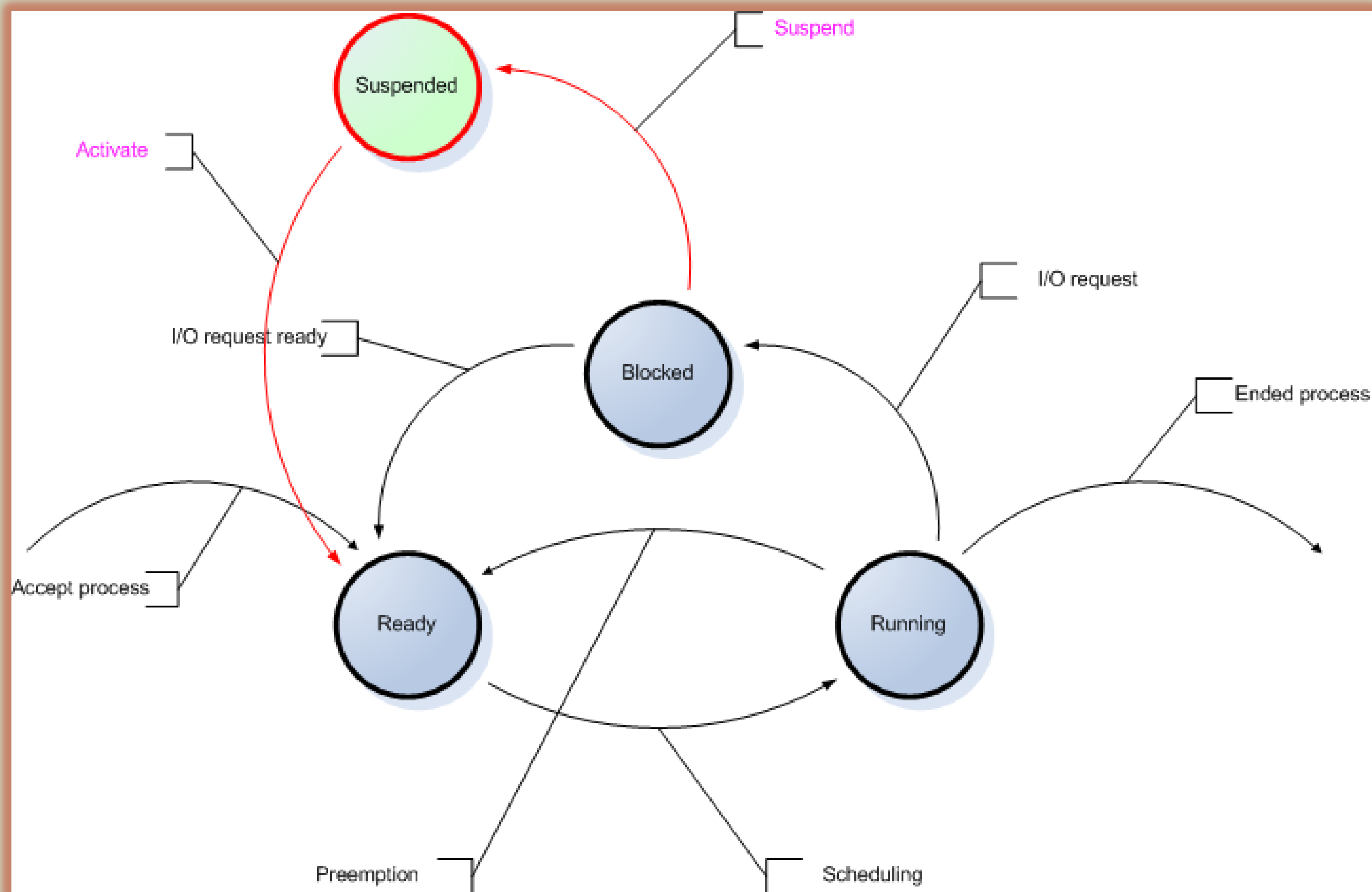
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

A process can be in running, blocked, or ready state.
Transitions between these states are as shown.

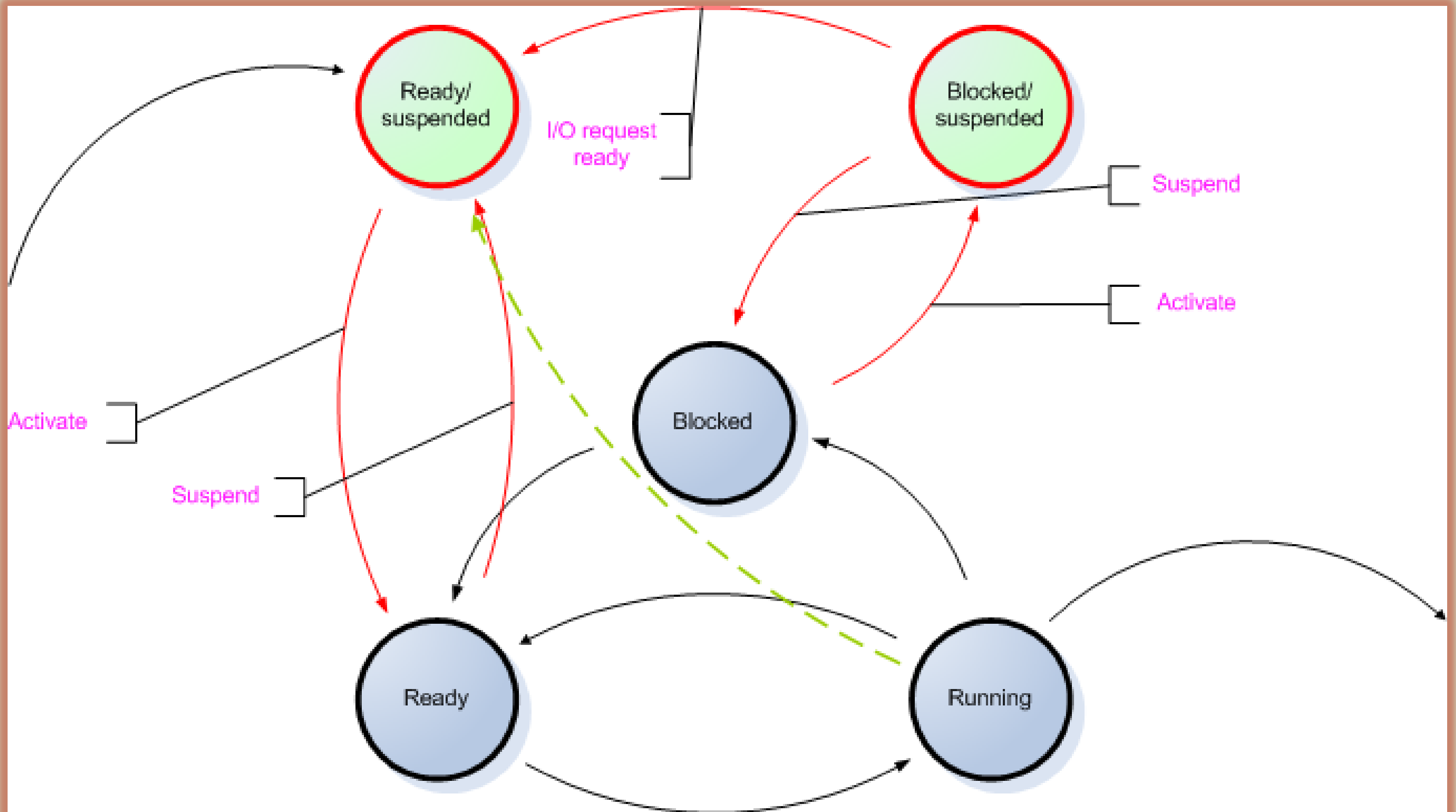
PROCESS' STATES
THE MODEL WITH ONE SUSPENDED STATE

- With the classical model one can to get an overview for process execution.
- As a result of virtual memory techniques, the model with 3-states could be modified to offer supplemental states in order to emphasize the swapping mechanism.
  - A blocked process (from memory) could be transferred on the disk, as a suspended process. Suspended process's activation can be done by loading it again in the memory as a "ready to run" process.
- Suspended processes should not be loaded into memory in another state unless they are scheduled for execution.

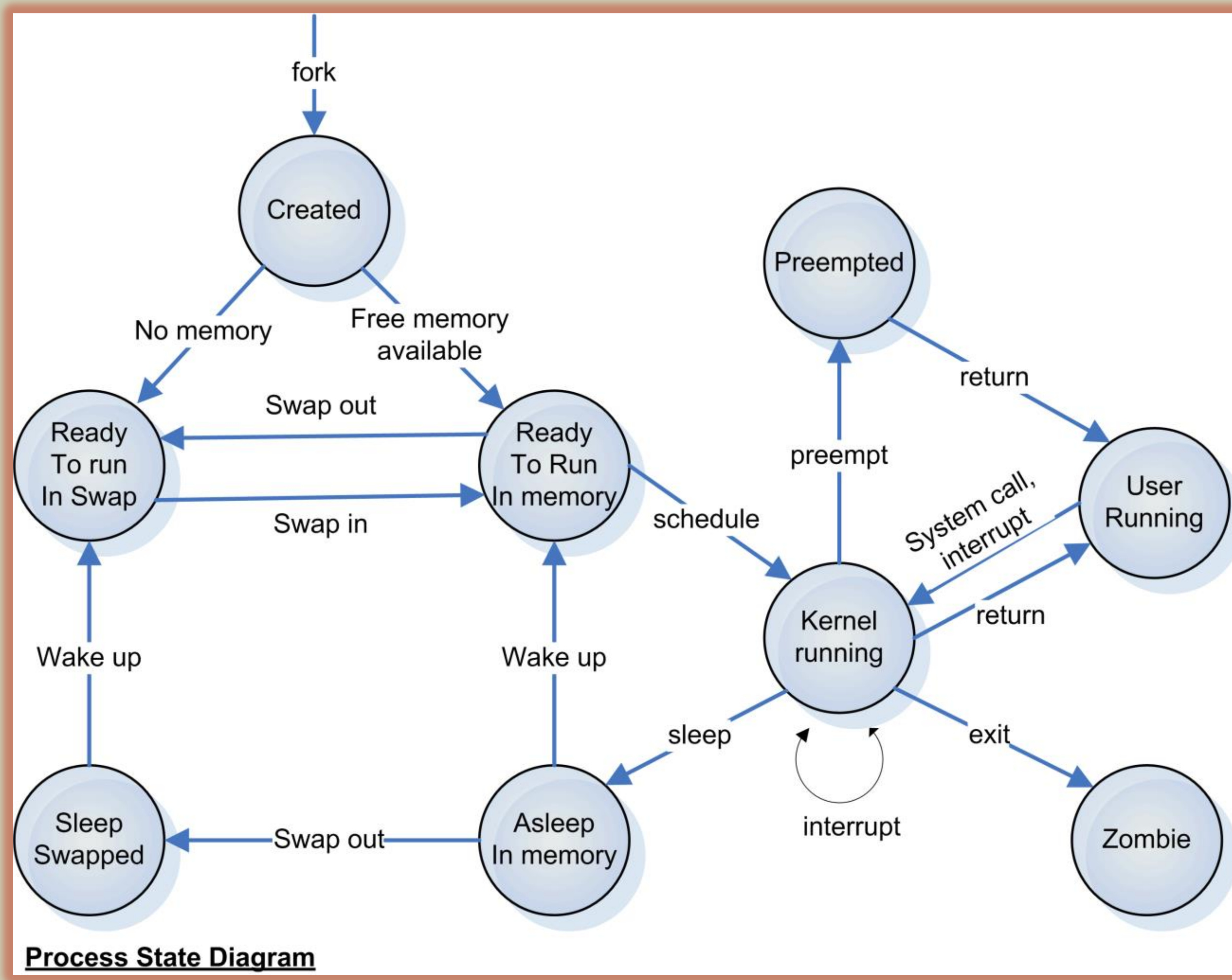
PROCESS' STATES
THE MODEL WITH ONE SUSPENDED STATE



PROCESS' STATES
THE MODEL WITH TWO SUSPENDED STATE



UNIX PROCESS' STATES



From http://www.ksvali.com

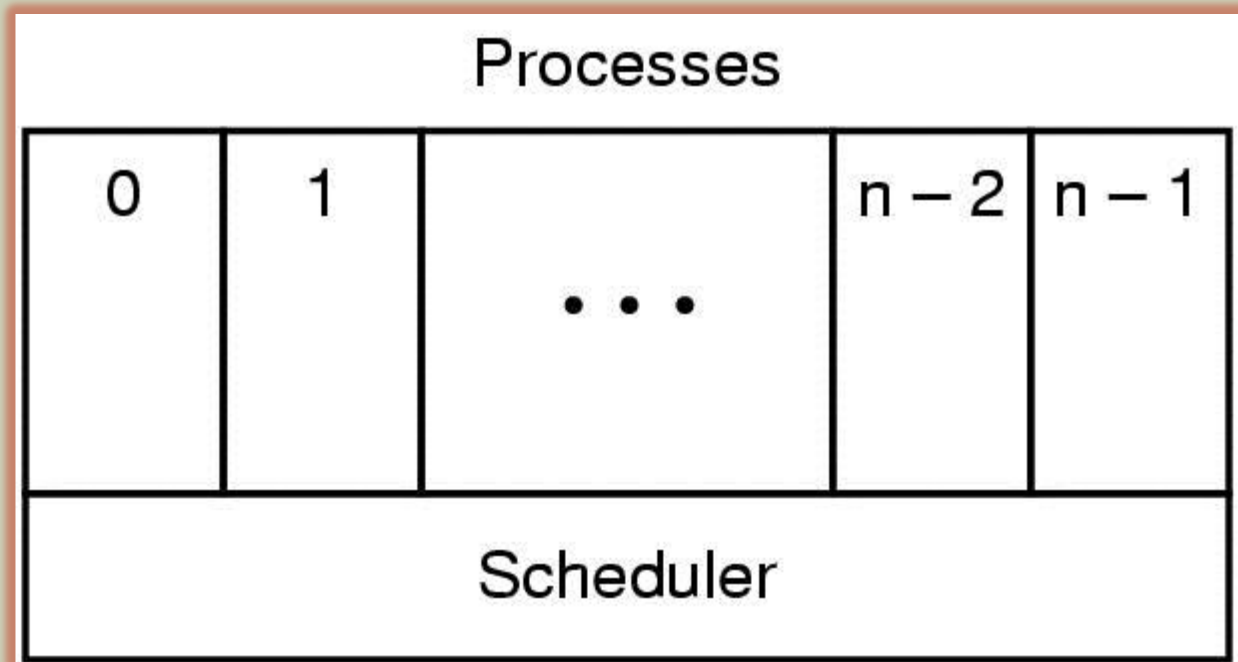
PROCESSES

Implementation of processes

PROCESSES
INFORMATION ABOUT PROCESSES

- The OS permanently keep in memory information about processes, in a specialized table (the process table).
- Each process in memory or virtual memory is characterized by an entry in this table.
- The information stored in this table form the PCB (Process Control Block). This includes information that define the context of a process:
  - Information for file management, such as: UID, GID, file descriptors, root directory, current (working) directory.
  - Memory information: pointers to the segments of the program.
  - Specific information: registers, scheduling information, PID, PPID, PGRP, signals, CPU time(s), and others.

PROCESSES
IMPLEMENTATION OF PROCESSES



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

PROCESSES
IMPLEMENTATION OF PROCESSES

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.

Skeleton of what the lowest level of the operating system does when an interrupt occurs.

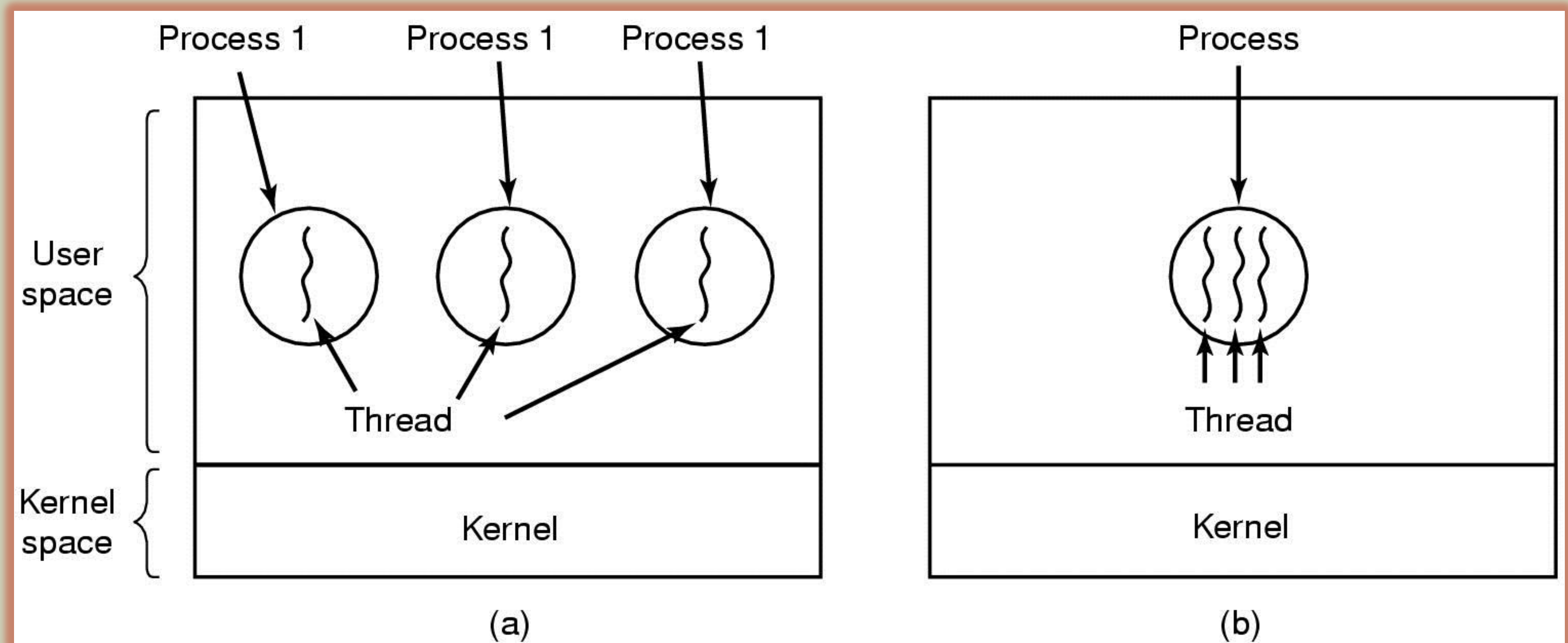
THREADS

Introduction

THREADS
INTRODUCTION. CLASSICAL THREAD MODEL

- Classical processes are characterized by a single execution, following only one thread (understood as direction) of execution. This single thread offers a simple, sequential execution model.
- Threads offer an extension to the classical model of processes, by offering parallel execution that are able to run in the space of a single process, these threads having also a certain degree of independence.
- Threads are using a limited quantity of information, as defined in the context of “hosting” process. This limitation refers only to information that are strictly related with current execution.

THREADS
INTRODUCTION. CLASSICAL THREAD MODEL

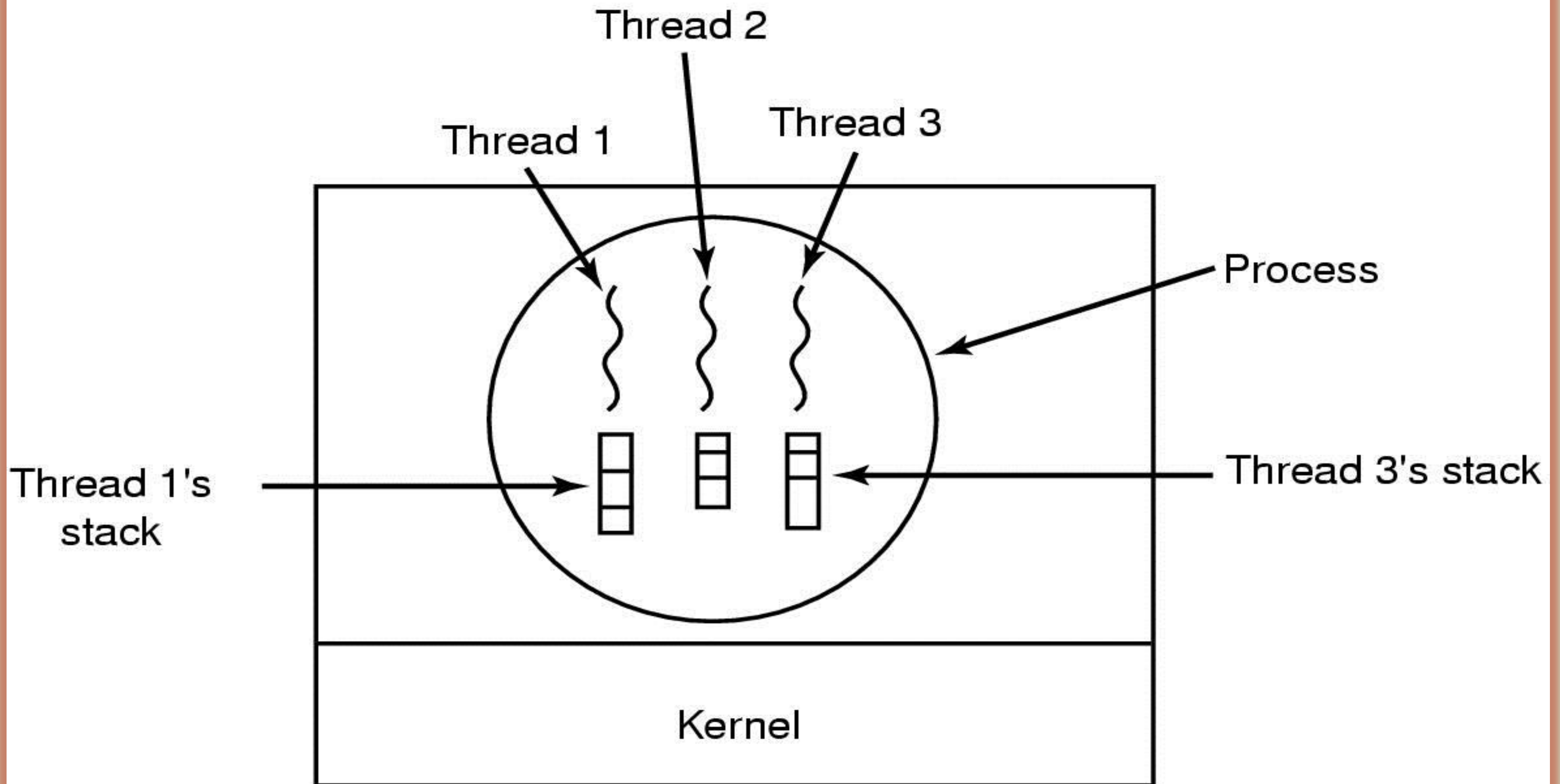


(a) Three processes each with one thread.
(b) One process with three threads.

THREADS
INTRODUCTION. CLASSICAL THREAD MODEL

- Threads are running in the context of a given process, inheriting several properties from this process. Sometimes you can see the term of "*lightweight process*" in order to designate threads.
- Threads are meant to offer the necessary means for executions, instead of managing resources that exist (and are shared) in the context of current process.
- Classical processes are highly oriented to resource identification and grouping, and also to execution.

THREADS
INTRODUCTION. CLASSICAL THREAD MODEL



Each thread has its own stack.

THREADS
INTRODUCTION. CLASSICAL THREAD MODEL

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	

The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

THREADS
INTRODUCTION. CLASSICAL THREAD MODEL

Items shared by all threads in a process

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

Items private to each thread

- Program counter
- Stack
- Registers
- State.

THREADS
INTRODUCTION

- There is a high *interdependence between the threads*. This is because threads are running in the same process space, and they have uniform access to every available resource from the "host" process.
- Any thread can interfere with the execution of any other thread.
- There is no protection mechanism at thread level. This kind of protection is not necessary, considering the main characteristic of threads:
  - To offer parallel execution in order to (collaborate for) solve problems in the space of the same process.

THREADS
INTRODUCTION

- Unlike multi-programming, *multi-threading* is based on the presumption that several threads of the same process could share several logical resources that are being used by the process.
  - Multi-threading does not necessarily require multi-tasking support from the operating system.
- Multi-threading is offered in a manner similar with multi-programming: alternate execution of different threads by a permanent switch between the different thread that are being executed.
  - The switch mechanisms can be coordinated by the operating system itself, or can be commanded by the process, depending on the thread model being used.

THREADS

Thread

states

THREADS
THREAD STATES

- For threads, one can imagine a similar execution model as for processes, with the same basic states and transitions:
  - Thread "ready for execution", every thread that is able to run, as soon as the thread is planned for execution.
  - Thread "in execution" (or running), is the only active thread, the thread that controls the current process.
  - "Blocked" threads, are threads that are waiting for some events. However, a thread can be blocked at the request of another thread.
  - "Ended" threads, are threads that have finished their activities, and are waiting to report execution results (by using the join mechanism).

THREADS
THREAD STATES

- In the case of the Windows 2000 model, there are six states:
  - **Ready**: This specify a thread that can be scheduled for execution. The micro-kernel should decide which of the ready threads is going to be scheduled.
  - **Standby**: This state specify a thread that has been selected to run on a processor. It is maintained in this state until the processor is ready (eventually, after preempting current thread).
  - **Running**: This is the active thread. Its execution ends if it terminates, it is blocked, preempted or its time slice expires.
  - **Waiting**: This is a thread waiting for I/O operation, a synchronization or it has been suspended.
  - **Transition**: This temporary state specify waiting threads whose resources are temporarily unavailable.
  - **Terminated**: a terminated thread, by one of the following reasons: normal termination, by another thread, by parent (process) termination.

THREADS

Implementation

THREADS
CREATING THREADS

- Each process is starting its execution as a single-threaded process.
- The main thread is responsible with the creation of other threads. Later, every other thread is also able to create other threads.
- Recalling threads characteristics, maintaining and using parent-child mechanisms is a difficult task.
- The typical call for this operation is of type `thread_create()`

THREADS
TERMINATING THREADS

- As for processes, thread termination can be realized explicitly, by a thread_exit() call, or implicitly, once the thread has returned and/or finished the execution of its "main" function.
- Threads offer their own synchronization mechanism, by using the basic thread_wait() and thread_join() calls. In order to use this kind of synchronization, a thread must be in a ***joinable*** state.
- Synchronization mechanisms could also include several OS primitives, as condition variables (used as a mechanism for signaling some continuation conditions), mutual exclusion (***mutex***) mechanisms, semaphores, and others.

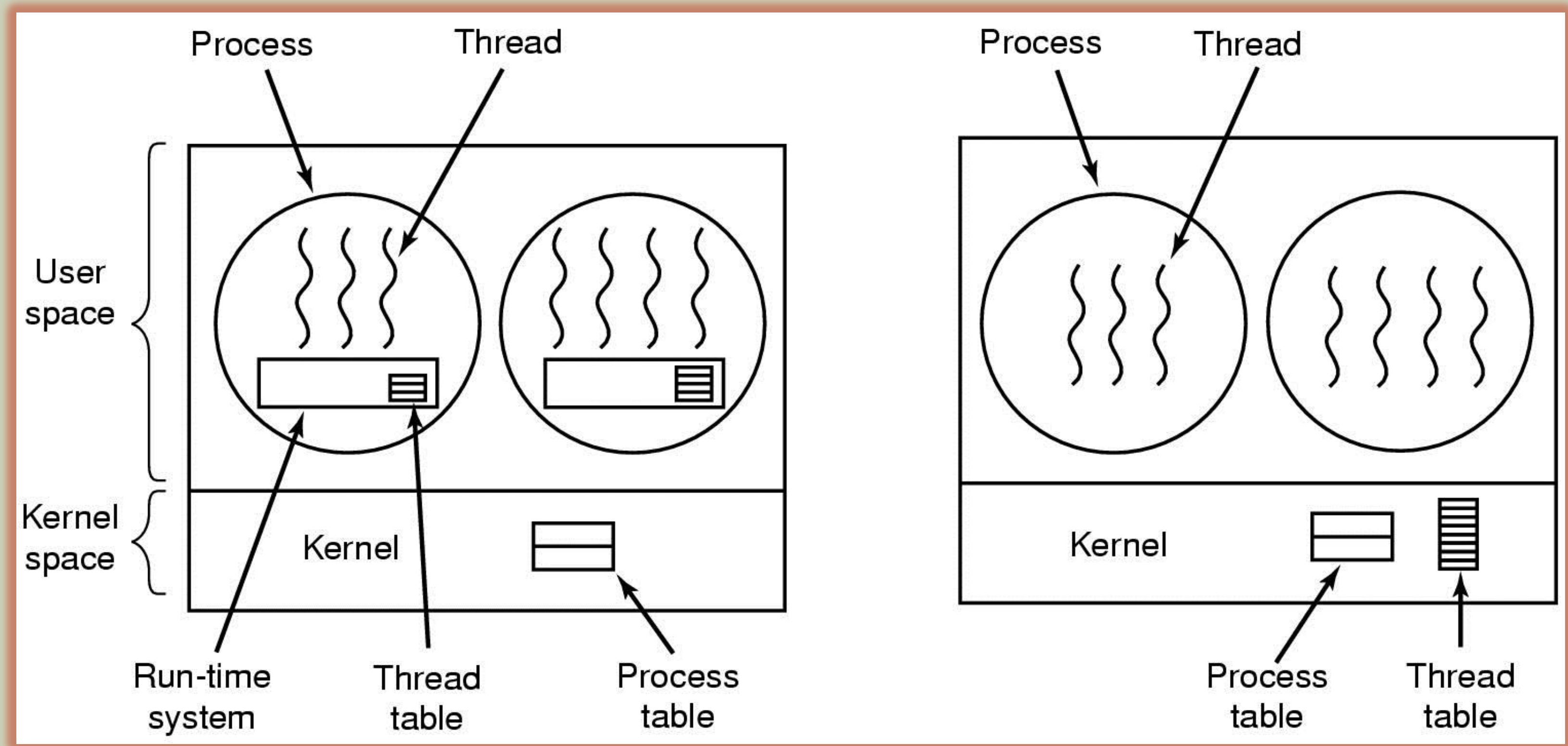
THREADS
EXECUTION CONTROL

- Threads are being executed in the space of a single process, so mechanisms for limiting execution time are no more effective.
  - Threads always compete for fulfilling some tasks in process space. Following this idea, it should be very important that threads are able to voluntarily offer process' resources to other threads.
- Threads are using the thread_yield() call in order to offer the control to another thread; this call could be used in order to evaluate some conditions.
- In order to device a thread control mechanism, one could consider several problems, like:
  - Inheriting threads from child processes (usually, this kind of facility is not offered by the OS)
  - (Re)using resources that are exclusively owned by a thread, after its unexpected end (for example, due to another thread).

THREADS
IMPLEMENTATION. THREADS IN USER SPACE

- An approach for simple operating systems, like mono-user OS, for an OS without the necessary support for threads, or for every other OS.
- Thread execution is made only in user space:
  - The OS does not know anything about these executions.
- The entire management is made at process level, by offering a specialized thread-management subsystem.
- The management is based on a “thread” table, being accompanied by decisions similar with scheduling decisions.

THREADS IMPLEMENTATION. THREADS IN USER SPACE



(a) A user-level threads package.
(b) A threads package managed by the kernel.

THREADS IMPLEMENTATION. THREADS IN USER SPACE

- Because all threads are executing in user space, the activity of a scheduler should be quite fast.
  - Cons: when using blocking system calls (like *open*, *read* and others) it is possible to block the entire application, not only the current thread.
  - Cons: only one thread can be executed at a moment. The execution of other threads is completely based on a fair execution of the current thread (for example, current thread should issue explicit thread_yield() calls in order to voluntarily give the control to other processes).

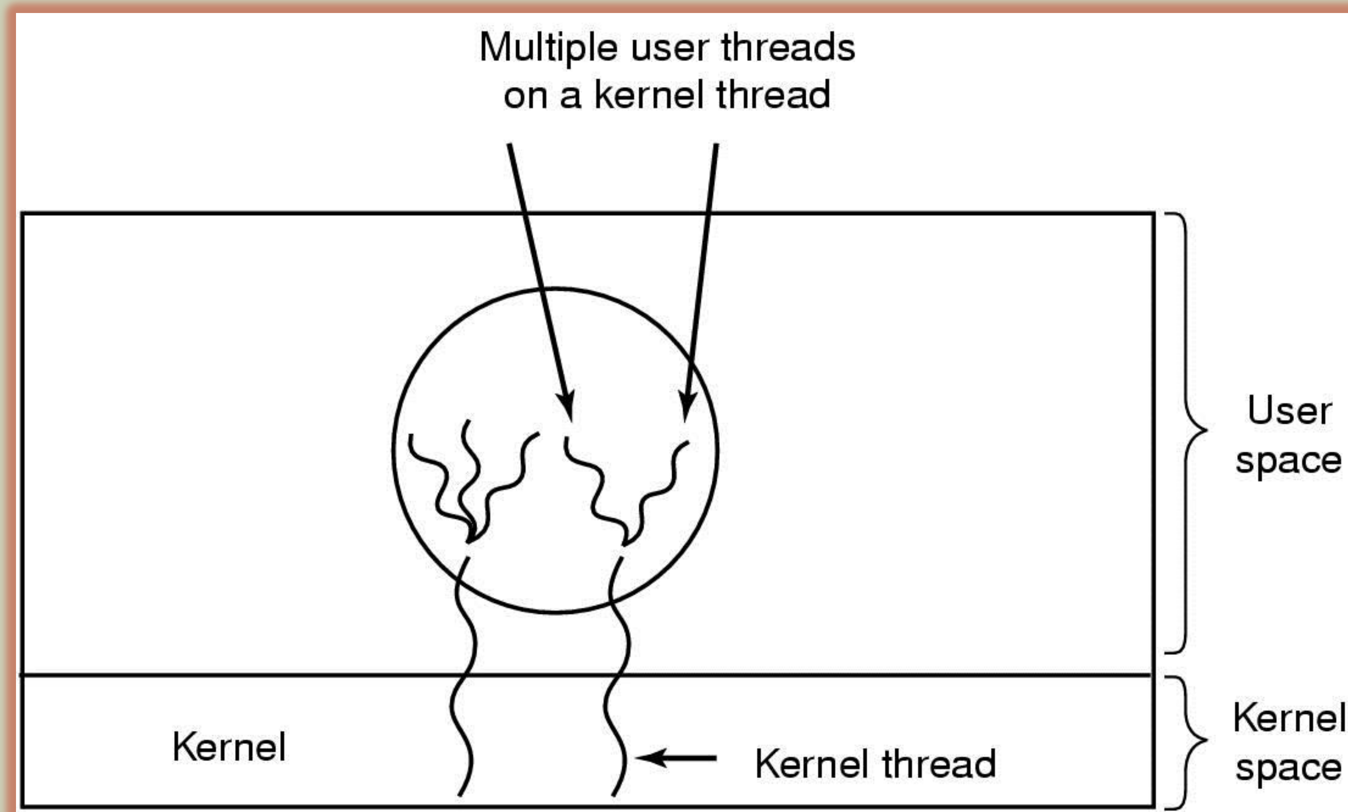
THREADS
IMPLEMENTATION. THREADS IN KERNEL SPACE

- The management activities for threads are “passed” to the operating system.
  - The OS should offer specific system calls for thread management, and should maintain supplemental information for thread management.
- Blocked threads does not necessarily block the entire application
  - The OS is able to schedule other threads from the same application, if necessary.
- Thread creation or deletion are now costly operations, similar with process creation or deletion.
  - In order to go over this difficulty, the OS could maintain a pool of thread-sockets, and deleted threads are only marked as being deleted; instead they are kept in the system in order to speed up future thread creations.

THREADS
IMPLEMENTATION. THE HYBRID APPROACH

- Usually the OS does not limit to one of the two implementations presented above. A hybrid approach is used in order to combine the advantages of user-space threads with the management for kernel-space threads.
- In the *hybrid approach*, threads are mapped in kernel space, that is able to offer typical mechanisms for thread management.
  - The OS kernel is only interested about these management activities, and not about the managed objects (that exist now in user space).

THREADS
IMPLEMENTATION. THE HYBRID APPROACH



Multiplexing user-level threads onto kernel-level threads.

THREADS
IMPLEMENTATION. COMMON PROBLEMS

- **Global variables** can rise severe problems when used in a multi-threaded application. These problems are due the fact that threads can access all the information in the space of the process.
- Because there are no clearly defined protection mechanisms, it could be possible that several threads that share the same global information could be able to simultaneously modify its value, thus offering the possibility of incorrect usage of these information.
  - For example, one thread execution is based on a value managed by another thread. If the first thread is using this value before the other thread has been able to modify it, it is possible to have a wrong behavior from the first thread.
- This problem can be solved as follows:
  - Avoiding the problem: do not use global data that can be changed.
  - Offering private views for global data (this kind of information can be named global-local variable)

THREADS
IMPLEMENTATION. COMMON PROBLEMS

- A library function is reentrant if a second call for this function is possible while its first call is still running (not necessarily from the same thread).
- Sometimes such a function is a … *thread-safe* function. Most of the library functions are not reentrant. The second call could live several internal structures in an inconsistent state.
- Solutions for this problem include:
  - Rewriting the library functions in order to offer reentrant variants;
  - Forbidding calls for unsafe (non-reentrant) functions.

THREADS
IMPLEMENTATION. COMMON PROBLEMS

- For OS with threads in user space, when a stack for a thread is full, the OS does not necessarily promptly react. This could be possible because there is no clear request from the process that own the thread.
- The process itself is responsible with stack management, so the thread-subsystem should solve supplemental requests from threads and redirect them to the OS.

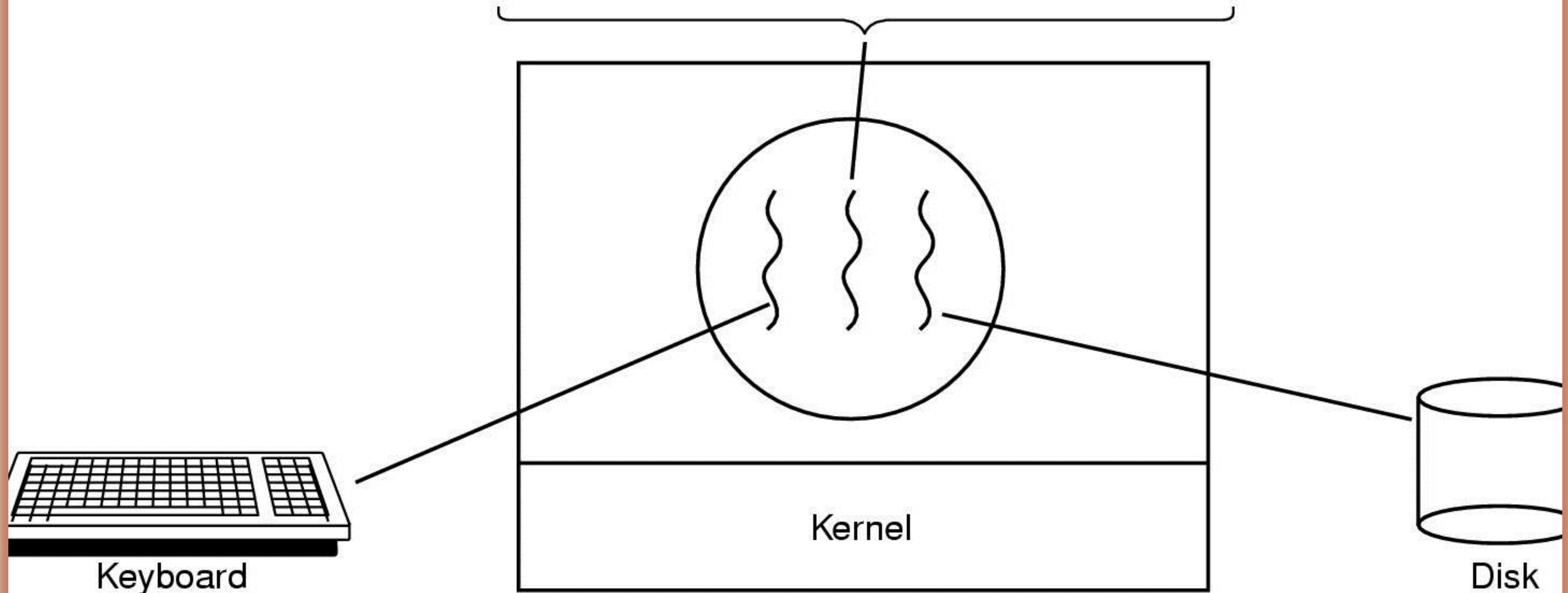
THREADS
USING THREADS

- There are several reasons for thread usage, including the following:
  - Foreground and background processing: a simple application could use multithreading in order to display menus, perform a document search, and an update activity at the same time.
  - Asynchronous processing: asynchronous activities could be implemented in a program as different threads. An update activity, a periodic save of a document can be considered asynchronous activities.

THREADS
USING THREADS. EXAMPLES

- Speed-up execution: threads can be used in order to overlap operations.
  - For example, one thread could make some computations, while another thread is preparing data for next computation.
  - If a problem can be decomposed and various sub-problems can be solved simultaneously, one can use several threads in order to perform these computations.

THREADS
USING THREADS. EXAMPLES

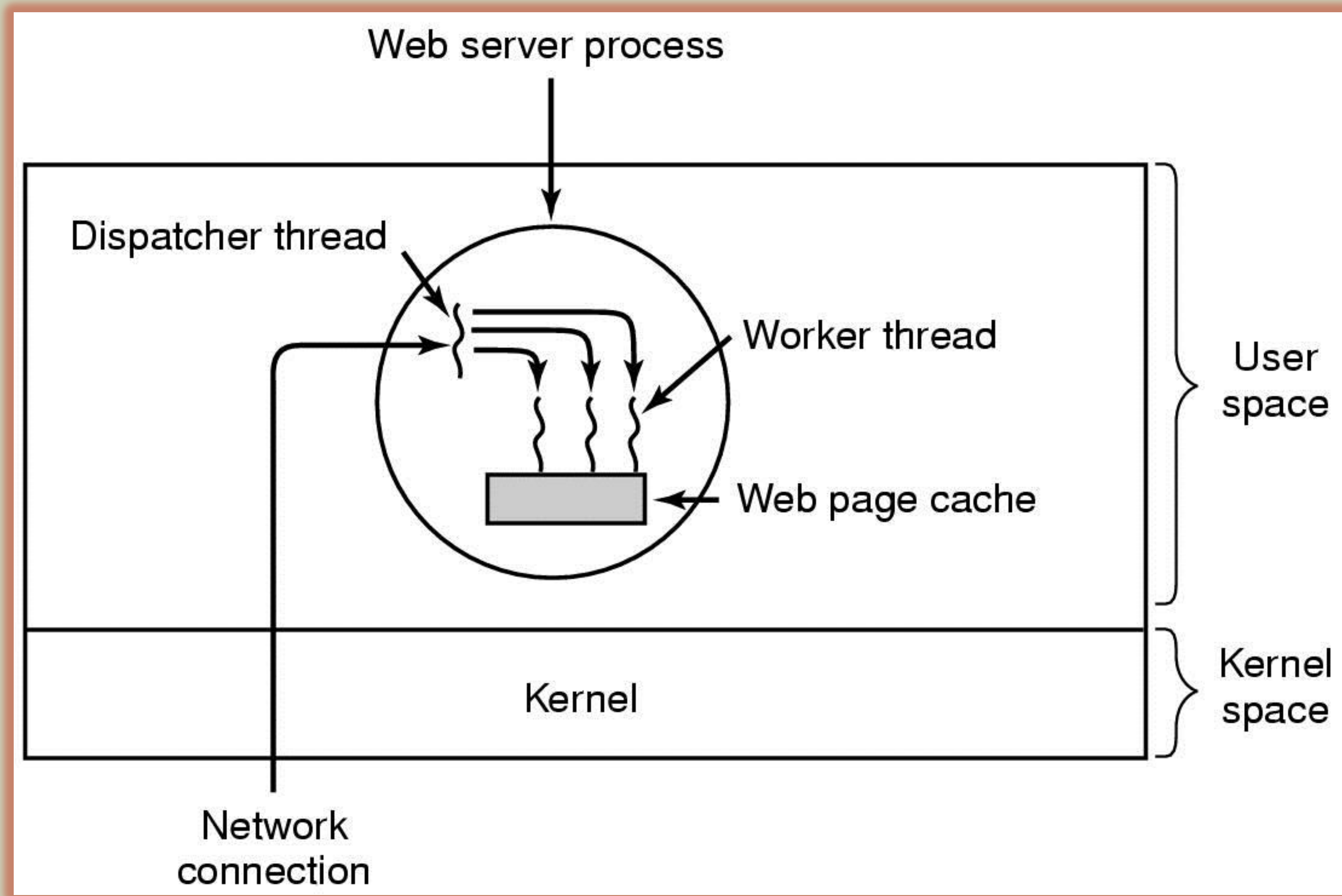


A word processor with three threads.

THREADS
USING THREADS. EXAMPLES

- Modular program structure:
  - Multi-threading can be used in order to offer an easier design and implementation for applications that perform numerous activities, with a large amount of sources and destinations for I/O operations.

THREADS
USING THREADS. EXAMPLES



A multithreaded Web server.

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

(a)

```
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);
}
```

(b)

A rough outline of the code for previous Figure
(a) Dispatcher thread.
(b) Worker thread.

THREADS

POSIX
Threads

THREADS
RECAPITULATION. MULTITHREADING

- Reasons for threads usage
  - High costs for creating new processes (via the fork() mechanism).
  - Threads require less memory at startup time.
  - Easy access to shared information / data.

- Each process is made up of one or several threads. Information shared by all threads of a process include:
  - Memory (as program code, global data);
  - Open file and socket descriptors;
  - Signal handlers and signal dispositions;
  - Environment information (like current directory, user ID, and others).

POSIX THREADS
THREADS AS LIGHTWEIGHT PROCESSES

- Like in the case of processes, each thread has its own information:
  - Thread ID (integer);
  - Thread stack, registers, program counter;
  - **errno** (if not - **errno** would be useless!)

- All threads within the same process have access to global data, as well as shared memory. Threads could use shared memory for communication purposes.

POSIX THREADS CHARACTERISTICS

- Widely supported threads API;
- Implementations available on various platforms, including Windows;
- For *NIX, and UNIX-like platforms, link your application against the POSIX threads library:
  - E.g. gcc … -lpthread
- Notice: compiler could use (on some systems) re-entrant version of existing libraries, if available.
  - E.g. strtok_r () is thread safe, while strtok () is not.

POSIX THREADS
THREAD CREATION

- Thread creation
  - Subject to the call of pthread_create function:

```
pthread_create(pthread_t *tid, const pthread_attr_t *attr,
     void *(*func)(void *), void *arg);
```

  - Function returns 0 for OK, a positive value in case of error (Notice: usually, C system calls return -1 in case of error)
  - The function *does not set* the *errno* value in case of error!
  - New thread ID is returned via the "**pthread_t *tid**" value.

POSIX THREADS
THREAD CREATION

- Thread creation
  - Specified **func ()** is the thread function to be called.
  - The thread will terminate once thread function is finished (there are some additional termination mechanisms, however).
  - The **pthread_attr_t * attr** pointer is used for thread attributes specification, including:
    - Information about detached state;
    - Scheduling policy.

    (Use NULL for default values of ‘attr’)

POSIX THREADS
THREAD CREATION

- Thread identification:
  - Thread IDs are unique in the context of current process. A thread can recover its ID via the **pthread_self ()** call.
  - Thread ID are of type '**pthread_t**'. There is nothing special: just an integer (unsigned) in disguise.

POSIX THREADS
THREAD CREATION

- On thread creation, one can pass arguments to the thread function, via the ‘**void * arg**’ parameter. Use *NULL* if there are no arguments.
- Default mechanism is using a simple ‘**void ***’ argument for the thread function.
- Complex parameter could be passed by using an appropriate package (e.g. struct, typedef):
  - create a structure and pass the address of this structure;
  - Notice: as threads are using different stacks (with private data), you cannot use some local variables in order to pass thread arguments.

POSIX THREADS
THREAD CREATION. AN EXAMPLE

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

void *print_num (void *a)
{
    int i, odd = *((int *) a), ret ;
    for (i=0; i<4096; i++)
    {
        printf ("%4d ", odd) ;
        odd += 2 ;
    }
    ret = (i*3) ;
    pthread_exit (&ret) ;
}
```

```
int main ()
    {
        int i, a=1, b=2, thrRet ;
        pthread_t thrID1, thrID2 ;
        pthread_create (&thrID1,
    NULL, &print_num, &a) ;
        pthread_create (&thrID2,
    NULL, &print_num, &b) ;
        pthread_join (thrID1, NULL)
    ;
        pthread_join (thrID2, (void
    *)&thrRet) ;
        printf ("\n%d\n", thrRet) ;
        return 0 ;
    }
```

POSIX THREADS
THREAD CREATION

- Thread lifecycle
  - The designed thread function '**func()**' is executed once a thread was created.
  - The thread will terminate its execution when the '**func()**' returns.
  - Alternatively, a thread could use the **pthread_exit()** call in order to terminate its execution.
  - However, a thread terminates when main function terminates or there is an explicit call for **exit()** from ANY thread from process space.

POSIX THREADS
THREAD STATES

- There are two states: *detached* and *joinable*.
- *Detached* thread
  - on thread termination all thread resources are released by the OS. A detached thread *cannot be joined*.
  - There is no mechanism to recover the value returned by thread function (in fact, there is no return value).
- Joinable thread
  - On thread termination the thread ID and exit status are saved by the OS.
  - A thread can join with another thread by an explicit call of pthread_join ():
    - Calling thread will wait (blocks) until specified thread terminates execution. However, the other thread must be in joinable state!

    int pthread_join(pthread_t tid, void **status);
  - Calling thread could recover 'exist status' of the other thread.

POSIX THREADS GLOBALS

- Problems with thread usage: global variables

```
atomic_t counter=0;
void *dummy(void *arg) {
  counter++;
  printf("Thread %u has counter %d\n",
    pthread_self(), counter);
}
void main() {
  int i; pthread_t tid;
  for (i=0;i<10;i++)
  pthread_create(&tid,NULL,dummy,NULL);
 }
```

- In this simple example, the 10 threads are able to access and alter simultaneously the ‘counter’ value. Unpredictable behavior could occur in real situation.

POSIX THREADS GLOBALS

- Problems with thread usage: global variables
  - When using some shared information (like accessing global variables), threads could use IPC techniques in order to solve the issues.
  - The pthread library offers support for the mechanism of Mutual Exclusion (mutex).
  - With mutex, different threads could control their access to shared information by using a lock.
  - Notice that this mechanism is not enforced by the OS!

POSIX THREADS
GLOBALS. MUTEX AND CONDITION VARIABLE

- A lock is implemented by a global variable of type pthread_mutex_t.
    - **pthread_mutex_t lock= PTHREAD_MUTEX_INITIALIZER;**
  - In the case of static variable, explicit initialization with PTHREAD_MUTEX_INITIALIZER is required.
  - More options for mutex initialization are available in the case of dynamic variables.
- Condition Variables
  - pthreads support *condition variables*, which allow one thread to wait (sleep) for an event generated by any other thread.
  - This allows us to avoid the *busy waiting* problem.
    - **pthread_cond_t foo = PTHREAD_COND_INITIALIZER;**

POSIX THREADS
MUTEX & CONDITION VARIABLES. USAGE

- **Using mutex.**
- The mutex mechanism offer two blocking functions:
  - *pthread_mutex_lock(pthread_mutex_t &);*
  - *pthread_mutex_unlock(pthread_mutex_t &);*
- Calling thread will block until it is able to finalize the specified action (mutex lock or unlock).
- **Using Condition Variables**
  - A condition variable is always used with a mutex.
    - *pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);*
    - *pthread_cond_signal(pthread_cond_t *cptr);*

POSIX THREADS
MUTEX & CONDITION VARIABLES. EXAMPLE

Standard "waiter"

```
pthread_mutex_t mutex;
pthread_cond_t cond;

pthread_mutex_lock (&mutex);
while ( !condition )
   pthread_cond_wait (&cond,
   &mutex);
do_something();
pthread_mutex_unlock
   (&mutex);
```

Standard "signaler"

```
pthread_mutex_lock (&mutex);
// make condition TRUE
pthread_mutex_unlock
   (&mutex);
pthread_cond_signal (&cond);
```