The University of Melbourne
School of Computing and Information Systems

COMP30023
Computer Systems

Semester 1, 2017

Process Management

Processes

A **process** is a program in execution.

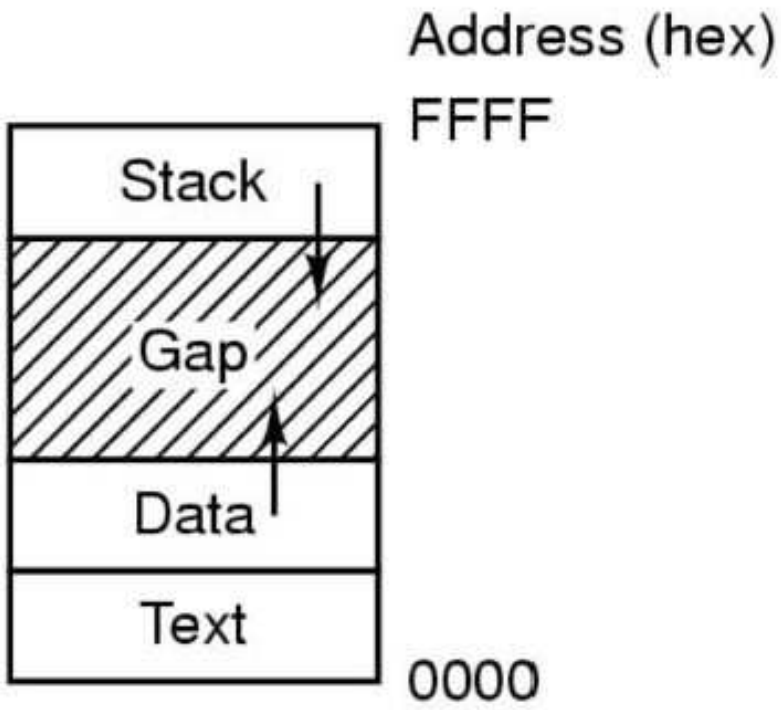
A **program** is static; a process is dynamic. At any given time, the number of processes running a given program can be 0, 1, 2, 3, ...

The state of a process consists of the code or "text" of the program, the values of all the variables, both in memory and in registers, the address of the current instruction, and probably some other data as well (e.g. current directory).

As an analogy, consider cooking. A recipe is a static entity; it is analogous to a program. A person cooking using that recipe is a dynamic entity; the act of cooking is analogous to a process.

Processes address space – a simple illustration

Processes have three segments: text (program), data (variables) and stack



User vs system processes

Most processes that a typical user deals with are created by that user (by invoking a program, either from the command line or via a GUI).

However, some services provided by the OS may also be implemented as separate processes. A typical example is a print daemon.

Try the command `ps -ef` to list all the processes running on a machine running Solaris or Linux.

Multiprogramming

Conceptually each process has its own virtual CPU.

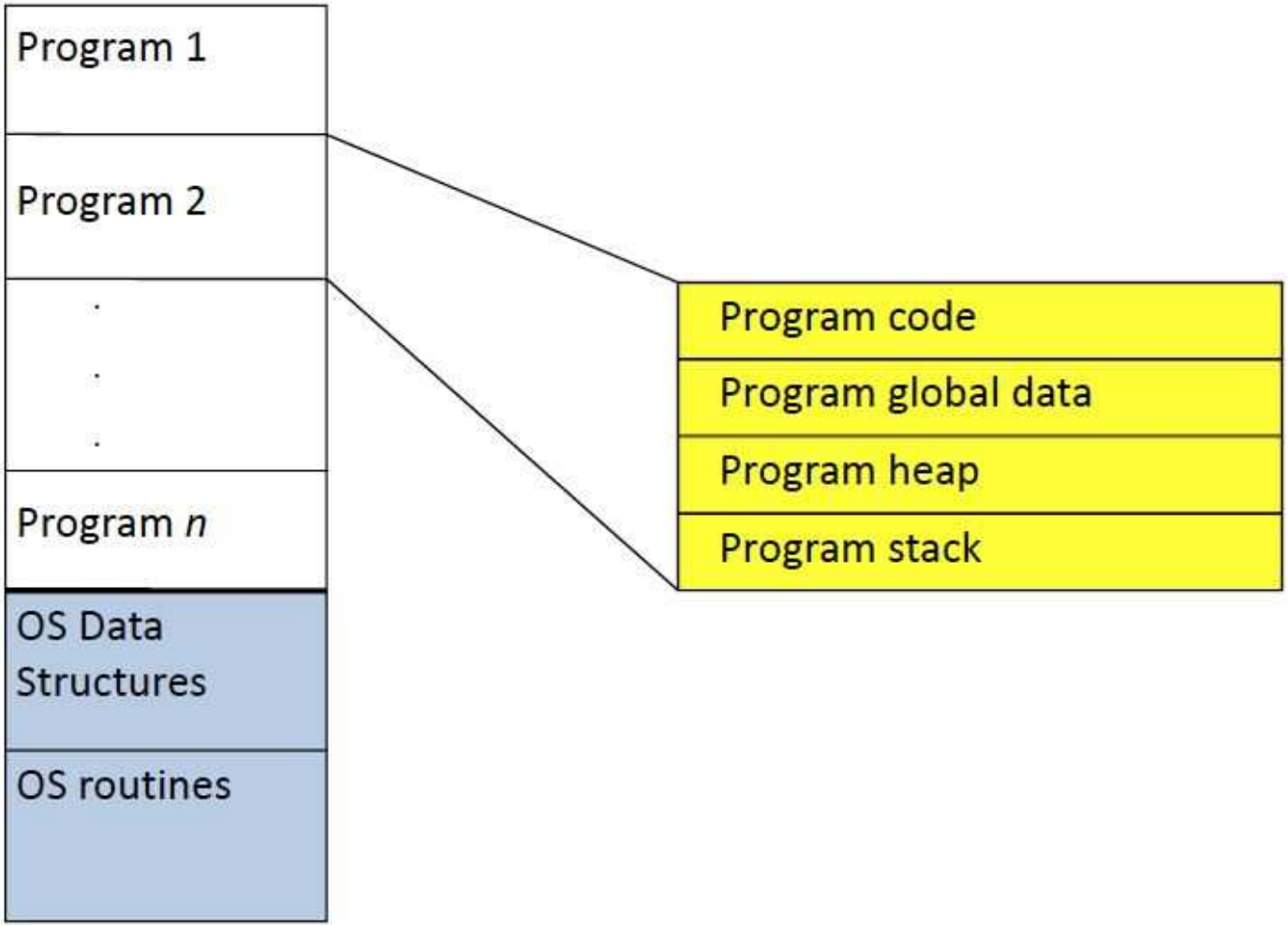
In reality, multiple processes will share a CPU, each running for a small period of time in turn. This is called **multiprogramming**.

Time-sharing is multiprogramming with simultaneous terminal access to the machine by several users.

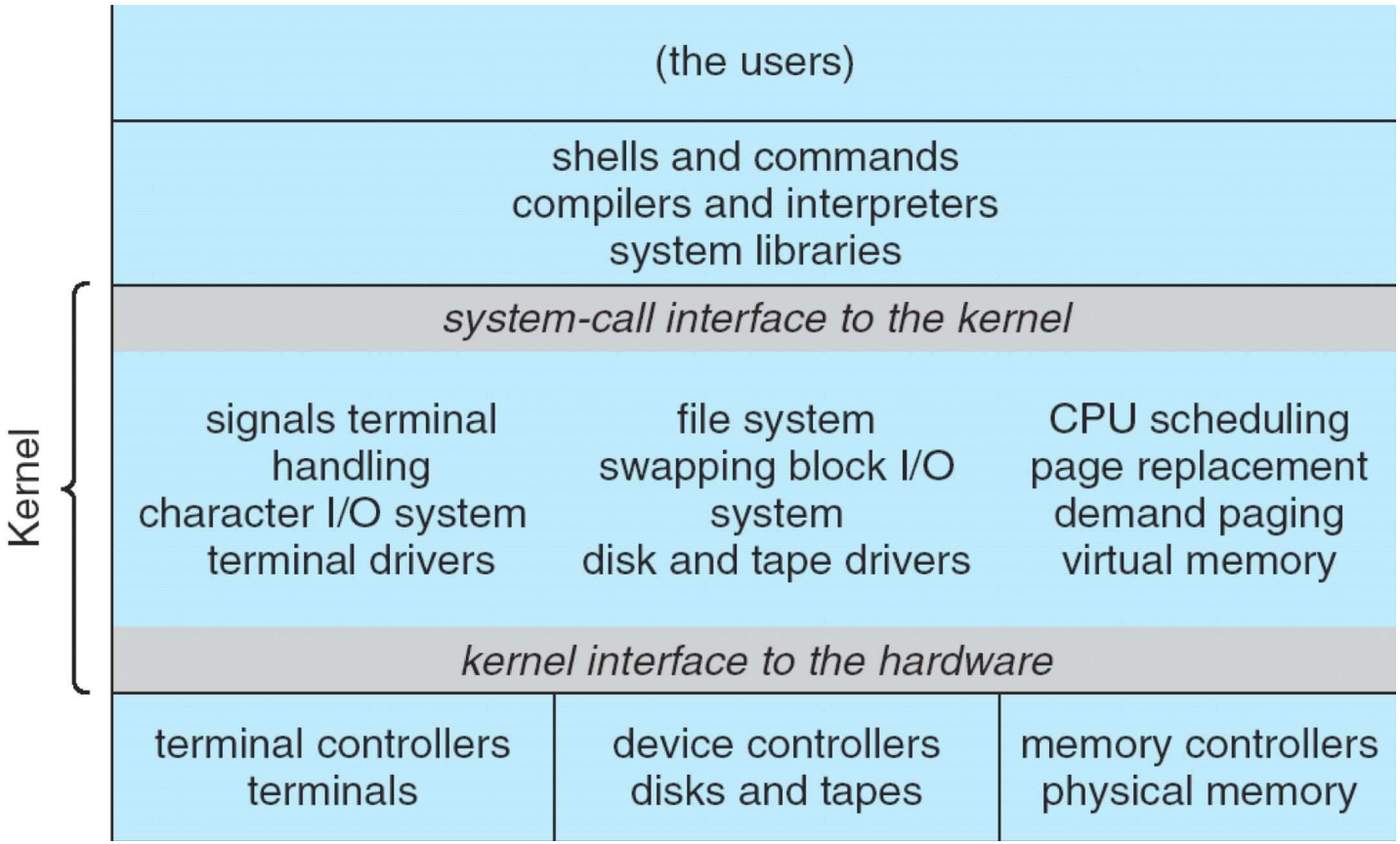
Multiprogramming increases system efficiency. When one process needs to wait for e.g. data from disk or keyboard input, another process can make use of the CPU.

Multiprogramming is useful even when the machine has two or more CPUs, since (for the foreseeable future) the number of processes will sometimes exceed the number of CPUs.

Multiple processes – a simple illustration



Unix system structure (an overview)



The kernel

If several processes are to be active at the "same" time, something has to ensure that they do not get in each other's way. That something is the privileged part of the operating system, usually called the **kernel**.

The kernel also provides services such as "read N bytes from this file". These services are used by application programs, utilities, and by the non-privileged parts of the operating system.

The kernel is not itself a process.

User-kernel distinction

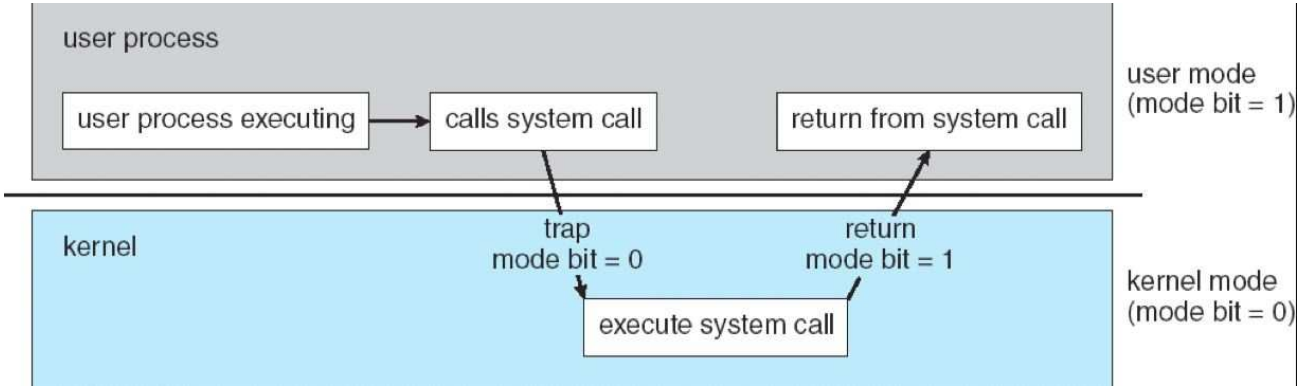
Most CPUs have two modes (some have more). The program status word (PSW) register gives the current mode.

- Code running in user mode cannot issue privileged instructions, and can access only the parts of memory allowed it by kernel mode code.
- Code running in kernel mode (also called system mode or supervisor mode) can issue all instructions, and can access all memory.

Instructions and memory locations whose use could interfere with other processes (e.g. by accessing I/O devices) are privileged.

The user mode / kernel mode distinction is the foundation needed by the kernel for the building of its security mechanisms.

Transition from User to Kernel mode



Mode bit (PSW) provided by hardware. It provides the ability to distinguish when system is running *user* code or *kernel* code.

System calls

System calls exist to allow user programs to ask the kernel to execute privileged instructions and access privileged memory locations on their behalf. Since the OS *checks* those requests before executing them, this scheme preserves system integrity and security.

To make system calls more convenient to use, a system call will typically do several privileged things in carrying out one logical operation, e.g. reading N bytes from a file on disk.

To the application programmer, a system call is a call to a privileged function.

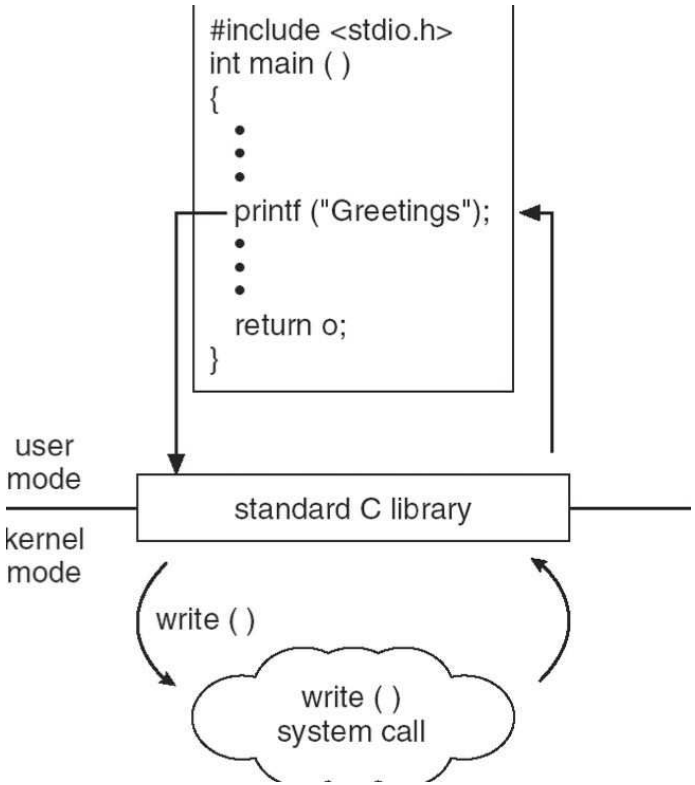
Example system calls on Unix:

- open, read, write, close
- fork, exec, exit, wait

Typically system calls (OS independent list)

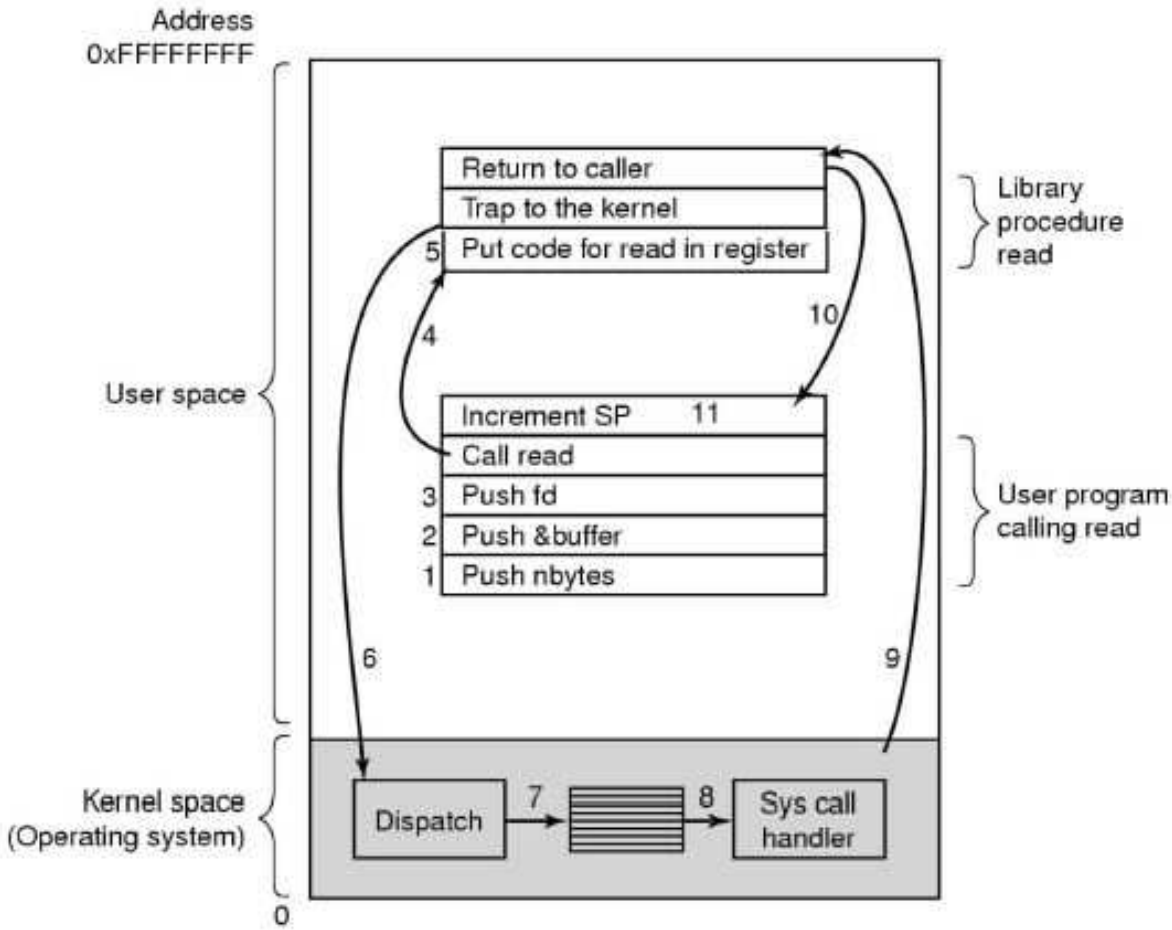
- **Process control**, eg. load/execute; create and terminate a process; get/set process attributes
- **File managent** eg. create/delete/open/close/read from a file; get/set file attributes
- **Device management** eg. request/release a device; read/write from a device
- **Information maintenance** eg. get/set time or date
- **Communication** eg. create/delete communication connections.

System call example

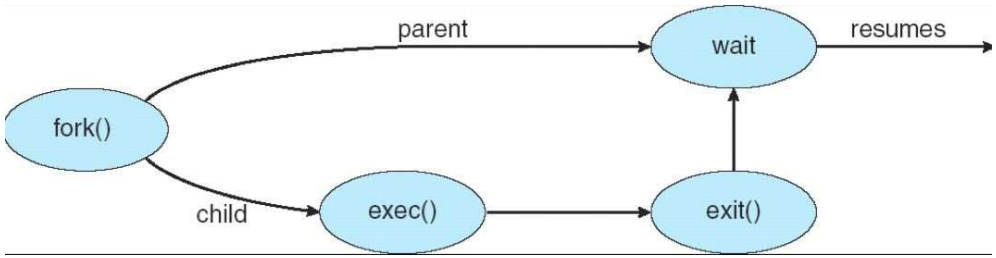


Standard C library handling of `write()`. The library provides a portion of the system-call interface for many versions of Unix and Linux.

Another system call example



Example: process creation / control



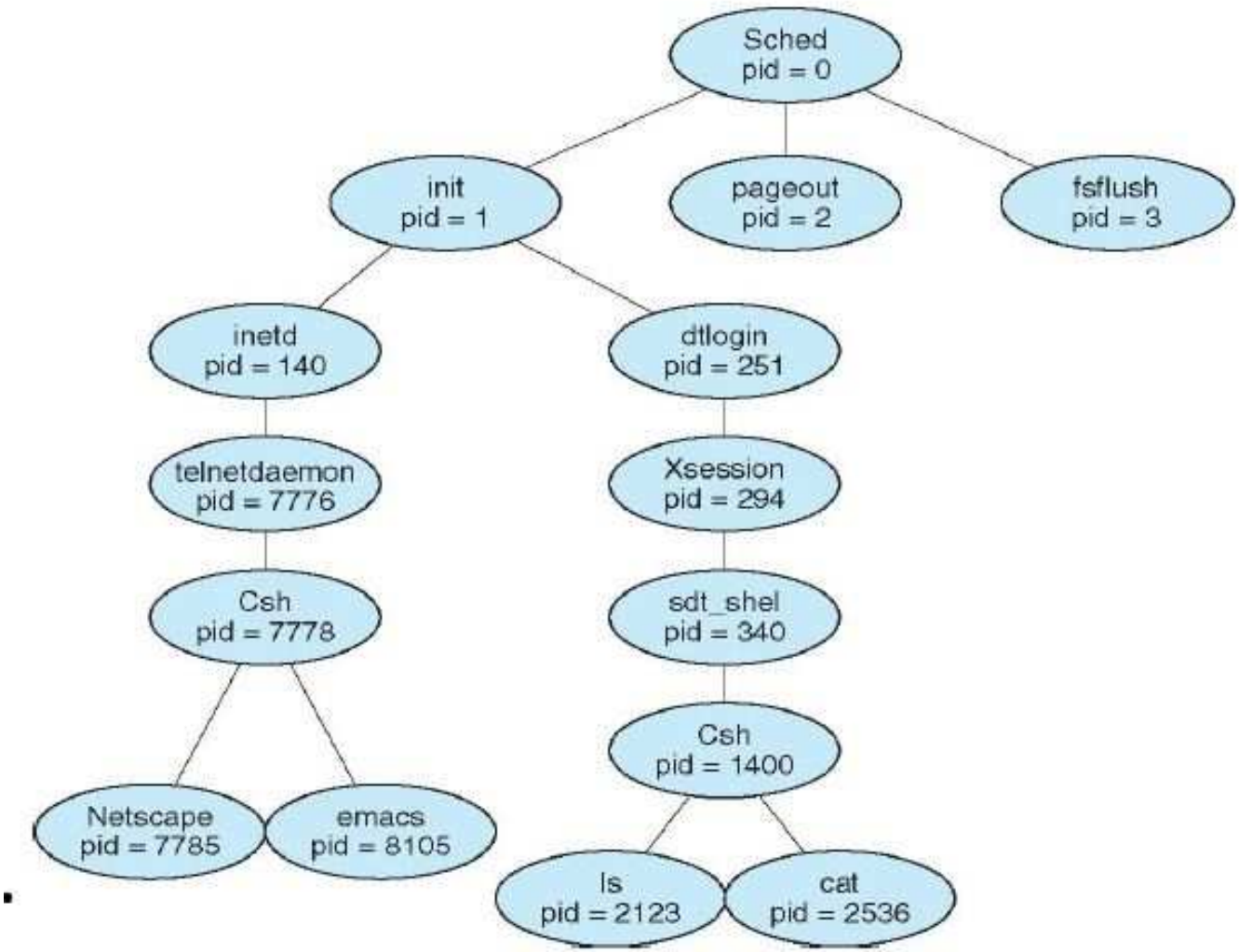
`fork()` creates a new process;

`execve()` is used after a fork to replace one of the two processess virtual memory space with a new program;

`exit()` terminates a process

A parent may wait for a child process to terminate: `wait` provides the process id of a terminated child so that the parent can tell which child terminated; `wait3` allows the parent to collect performance statistics about the child

Example: Unix process creation hierarchy



Interrupts

When a hardware device needs attention from the CPU, e.g. because it has finished carrying out its current command and is ready to receive its next command, it generates a signal to interrupt the CPU.

When an interrupt occurs, the CPU's hardware takes the values in the the program counter and program status word registers (and, on some kinds of machines, the stack pointer register), and saves them in privileged memory locations reserved for this purpose.

It then replaces them with new values.

The replacement PSW will put the CPU into kernel mode. The replacement PC will cause execution to resume at the start of the interrupt handler, code that is part of the kernel.

Interrupt handler

The interrupt handler must

- save the rest of the status of the current process,
- service the interrupt,
- restore what it saved, and
- execute a *return from interrupt* or similar instruction to restore whatever the hardware saved when the interrupt occurred (i.e. the PC, the PSW and (on some kinds of machines) the SP).

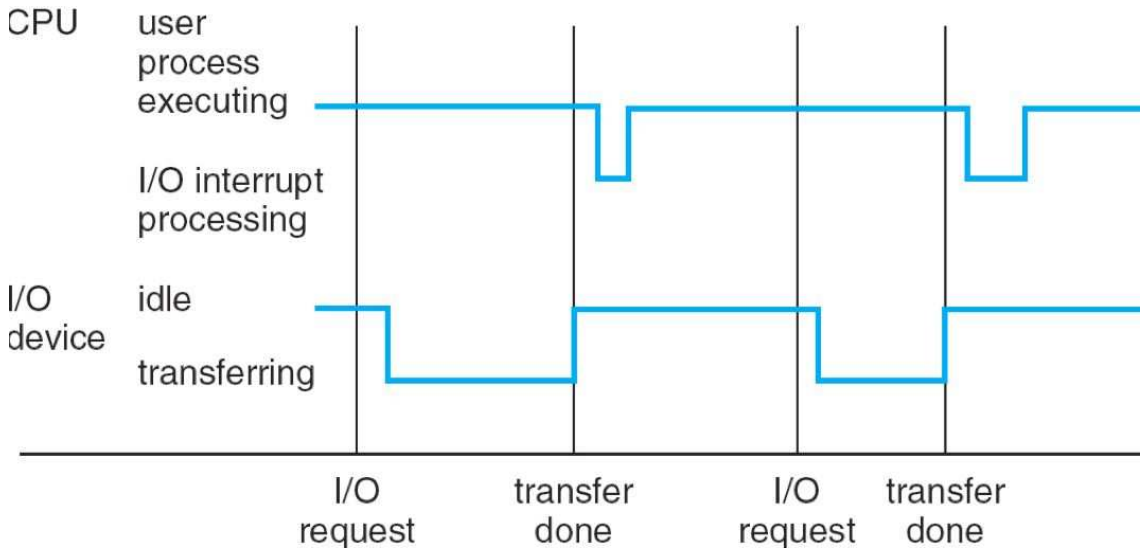
Interrupt vector

The interrupt vector is an area of memory that contains replacements for the program counter, program status word and (on some kinds of machines) the stack pointer for use when an interrupt occurs. The address of the interrupt vector is wired into the CPU.

The hardware interrupt signal specifies its source (i.e. what device generated it). The CPU hardware uses this information to select the proper entry in the interrupt vector. There is typically one such entry per device or class of devices.

The interrupt vector should not be directly writeable by users, since a user writeable interrupt vector is a security risk.

Example: Interrupt time line



An example interrupt time line for a single process doing output.

Interrupt priority level

Some interrupts need attention more urgently than others. The hardware associates a priority level with each type of interrupt.

The PSW contains the current IPL. The CPU hardware doesn't respond to a new interrupt unless the current IPL in the PSW is lower than the IPL of the interrupt.

Different machines have different numbers of priority levels. The Motorola 680x0 has 8, the Intel x86 has 32. The IPL is zero when a user process is running, so *any* interrupt will be handled then.

The OS executes the interrupt handler at the IPL of the interrupt being handled. This is done by loading the PSW from the interrupt vector at the start and restoring the saved PSW at the end.

Typical interrupt priority levels

A typical list of priorities would put the clock at the top, followed by networks, tapes and then disks.

The reason is as follows. Lost clock interrupts cannot be recovered, and clock interrupts are rare (60 a second) and cheap to handle.

Networks are fast, but recovering from a lost interrupt for a received packet requires the remote computer to resend its message, and this typically requires a timeout, which is slow.

Disks and tapes are local devices, and their status can be queried directly, i.e. without a timeout.

Pseudo-interrupts

True interrupts come from hardware devices outside the CPU, pseudo-interrupts from the CPU itself.

User programs may generate pseudo-interrupts inadvertently, e.g. by executing divide-by-zero: such events are usually called *exceptions*. Some exceptions cause process termination.

Users can generate pseudo-interrupts intentionally by executing a special instruction for system calls, e.g. TRAP in the 68000 and INT 2E in the Intel x86.

Each type of pseudo-interrupt has its own entry in the interrupt vector.

Summary: System call invocation

User programs invoke system calls like they invoke other functions. The code of the function is part of a standard library linked into the program. It is typically a short sequence of assembler instructions, which puts the syscall number and the arguments into the registers or memory locations where the kernel expects them, executes the syscall instruction (TRAP or INT 2E or ...), and returns.

The syscall instruction causes a pseudo-interrupt. The syscall entry in the interrupt vector causes execution to resume in the system call handler, which looks at the system call number argument, invokes the appropriate kernel procedure, and then executes the return from interrupt instruction.

This causes execution to resume in user mode at the instruction just after the TRAP (or INT 2E, or ...) instruction.

Summary: Interrupts/System calls/Traps

A system call instruction causes a **synchronous** exception (or *trap*).

Other sources of synchronous exceptions include: Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access); Segmentation Fault (address out of range); Page Fault (for illusion of infinite-sized memory)

Interrupts are **asynchronous** exceptions. Examples include: timer, disk ready, network, etc.

On a system call, exception, or interrupt, the hardware/OS enters kernel mode with interrupts disabled; saves PC, then jumps to appropriate handler in kernel.

System: Security

User programs execute in user mode, and cannot change to kernel mode without going through a system call.

A system call executes in kernel mode, but its first task is to check whether the user is authorized to perform the requested operation. It can be trusted to do this as it was written by the OS designer, and it is in memory inaccessible to users.

Users cannot bypass the authorization check (e.g. by causing kernel mode execution to start after the check) because that requires the ability to modify either the interrupt vector or the code it points to, both of which are in privileged memory.

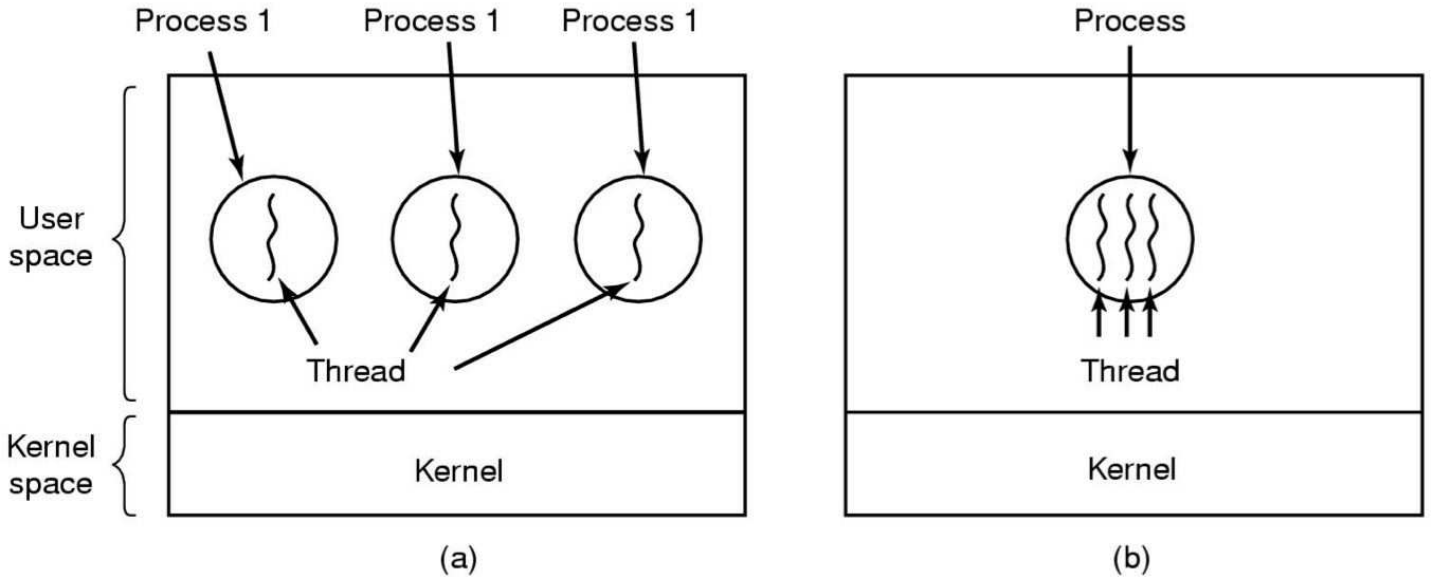
Threads

Some OSs provide similar functionality for sharing by dividing the notion of a *process* into two components: a thread, and a container for the thread.

Thread definition: a sequential execution stream within the process (sometimes called a "lightweight process"). Threads are the basic unit of CPU utilization – including the program counter, register set and stack space.

A process has one container but may have more than one thread, and each thread can perform computations (almost) independently of the other threads in the process.

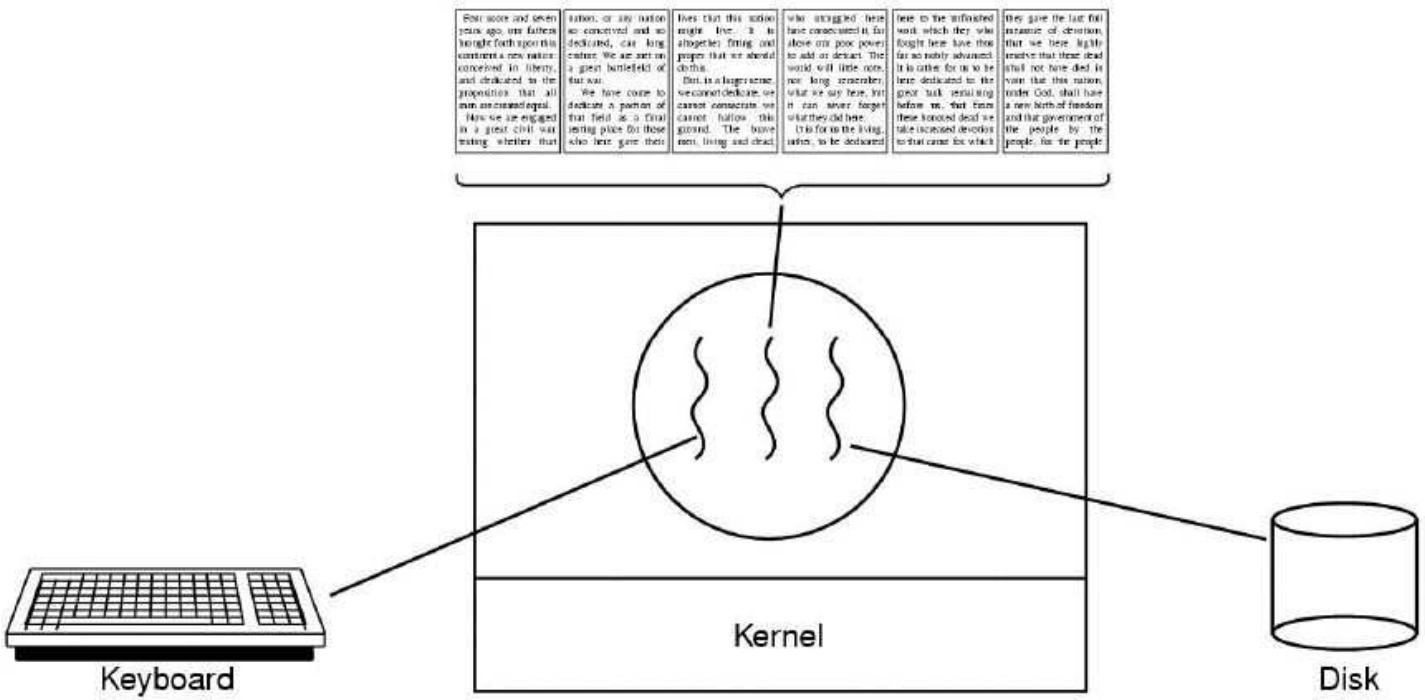
Threads



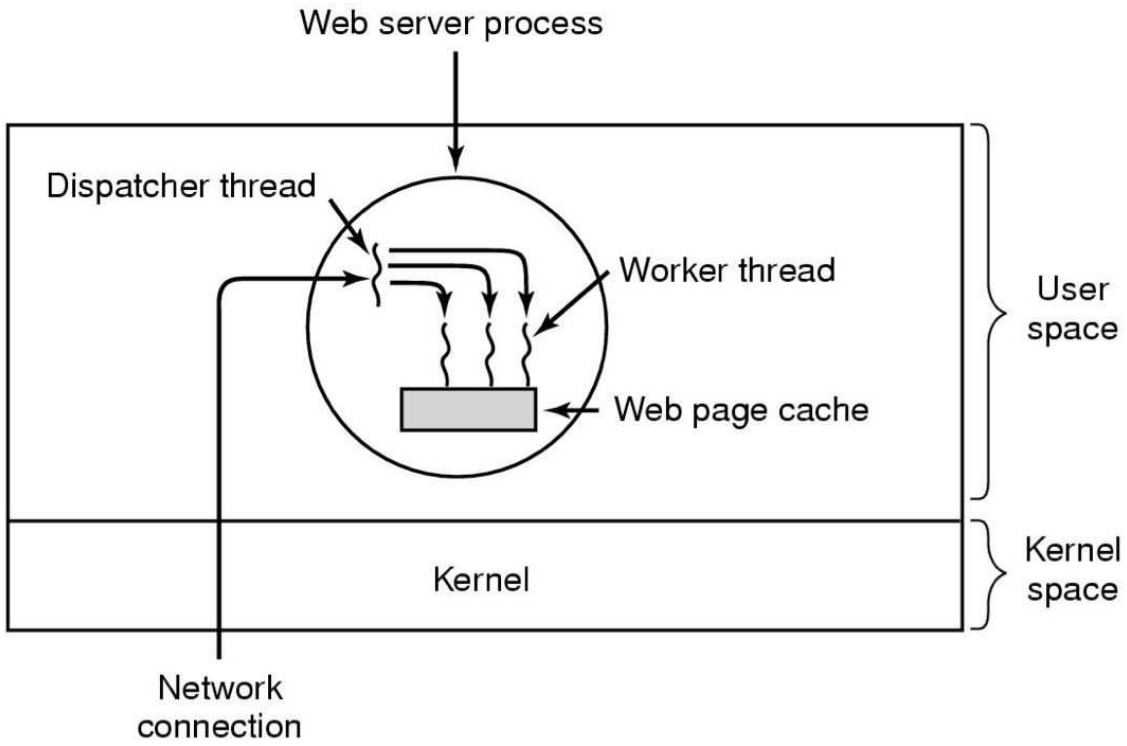
(a) Three processes each with one thread
(b) One process with three threads.

Threads share the CPU – only one thread can run at a time.

Example application – word processor



Application: Multi-threaded webserver



Application: Multi-threaded webserver

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

(a)

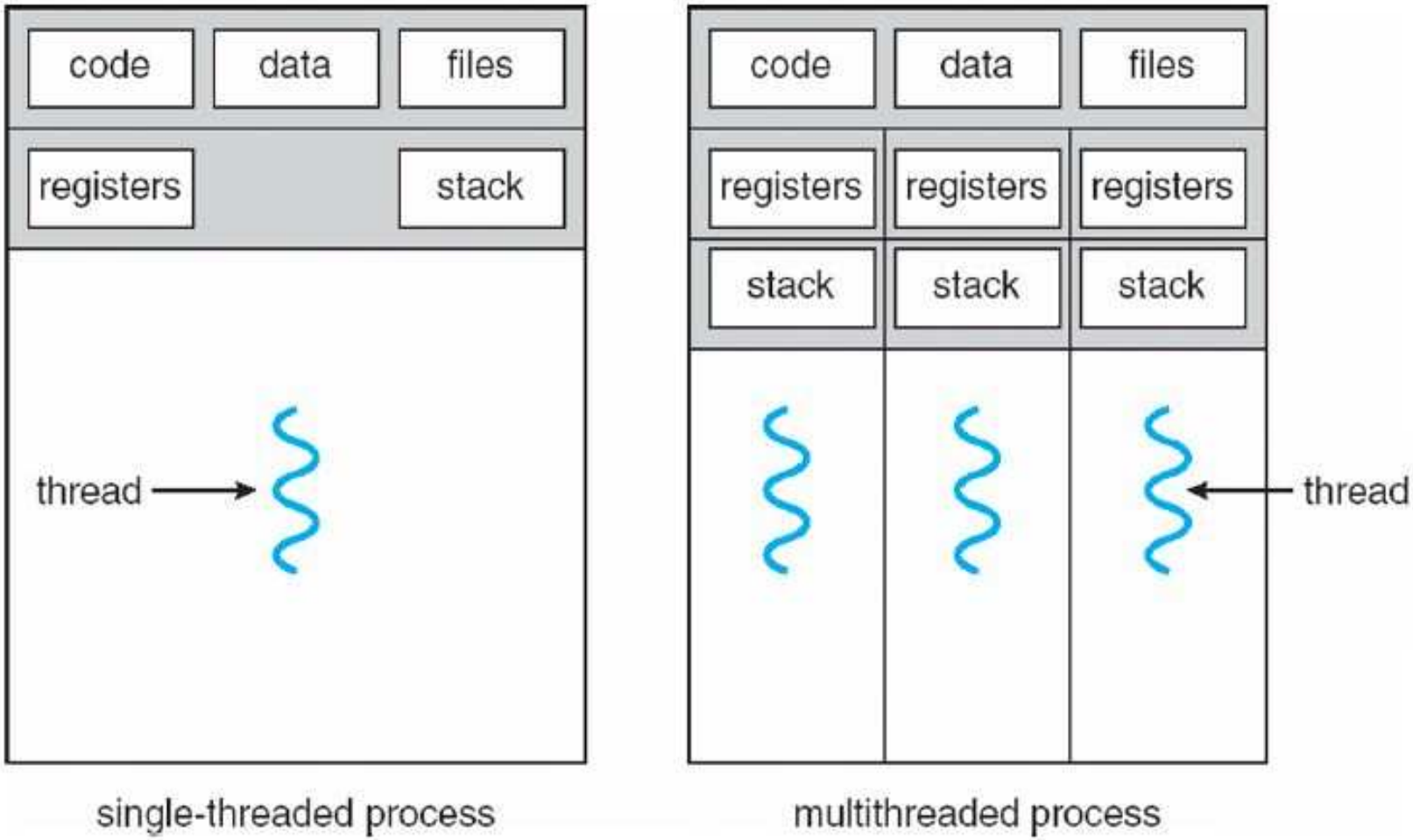
```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

High level outline of code for previous slide:
(a) Dispatcher thread
(b) Worker thread

. . . more details in weeks 9 and 10

Threads


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Threads

Shared by threads:

- address space and memory – code and data sections; contents of memory (global variables, heap) ; open files; child processes; signal and signal handlers;

Threads own copy:

- program counter; registers; stack (local variables, function call stack); state (running/waiting etc).

Threads can communicate with each other without invoking the kernel – threads share global valuables and dynamic memory. There is reduced overhead than when using multiple processes – less time to create a new thread; less time to terminate; less time to switch between threads; less time to communicate between threads.

Threads: processing system calls and interrupts

For servicing system calls and exceptions, the system switches to a preallocated *kernel stack* in system space. There is one such stack per thread.

For servicing interrupts, the system switches to a preallocated *interrupt stack* in system space. There is usually one such stack for the whole system, although there may be one per interrupt priority.

Threads

Solaris provides support for kernel and user level threads, symmetric multiprocessing and real-time scheduling. Note: the scheduler schedules threads, not processes.

This means that threads include at least the PC, PSW, the other registers, a stack, and the scheduling state (blocked/ready/running, and the current priority level).

What other process table fields are included in threads depends on the OS, but is fixed for a given OS.

In OSs whose kernels do not support threads, it is possible to simulate them in user level code, but only imperfectly.

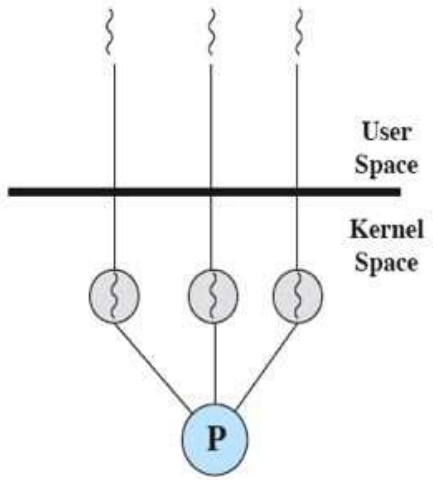
The main limitation of user level threads is that if one thread blocks, the entire process blocks, including all the other threads.

Kernel threads

Kernel level threads provide a one-to-one relationship (thread-to-process). The kernel maintains context information for the process and for individual threads within the process

Advantages: threads from the same process can still run if one is blocked; can take advantage of multiple processors

Disadvantage: overhead of creating kernel threads; a mode switch is required; less control over scheduling

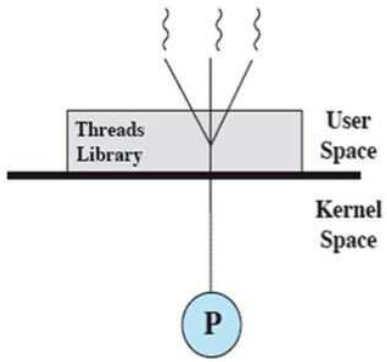


User threads

User threads are managed by a user thread library; runs in the user space of a proces. They provide a many-to-one relationship (between threads-to-process).

Advantages: mode switch not required; thread scheduling can be application specific

Disadvantages: system calls block threads – if one thread blocks, the entire process blocks including all the other threads; cannot take advantage of multiprocessors



(a) Pure user-level

Processes and threads in Linux

Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.

A distinction is only made when a new thread is created by the clone system call.

`fork()` creates a new process with its own entirely new process context

`clone()` creates a new process with its own identity, but that is allowed to share the data structures of its parent

Using clone() gives an application fine-grained control over exactly what is shared between two threads.

Pthreads

A POSIX standard API for thread creation and synchronisation. Common in UNIX flavour OS (Solaris, Linux, Mac OS X)

. . . see Pthread code examples

- all functions start with `pthread_`
- include `pthread.h`
- all threads have an id of type `pthread_t`
- need to use `-lpthread` when linking

Pthreads

Creating a thread

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

Id of thread itself:

```
pthread_t pthread_self();
```

Pthreads

Terminate a thread:

```
void pthread_exit(void *rval_ptr);

int pthread_cancel(pthread_t tid);

int pthread_join(pthread_t tid, void **rval_ptr);
```

Causes the current thread to wait until the specified thread has terminated.

```
int pthread_detach(pthread_t tid);
```

Causes the thread resources to be reclaimed immediately upon termination.

Pthreads

Global variables are shared across threads.

- thread switches could occur at any point
- thus, another thread could modify shared data at any time
- consequently, there is a need to synchronize threads – if not, problems could arise.