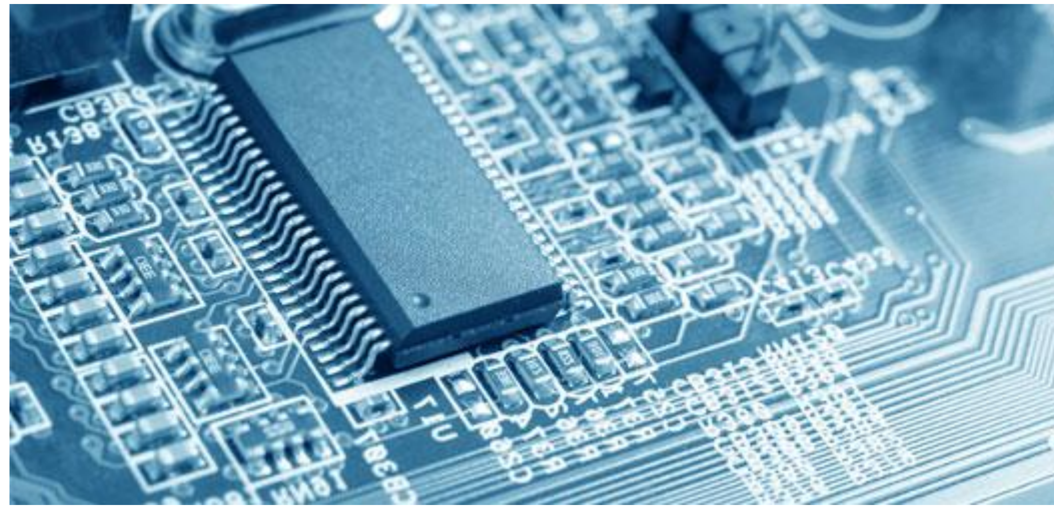
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



CSCI502 – Hardware/Software Co-Design

Lecture 4 – Real-Time Concurrent Programming Principles. Processes in Linux

28 January 2019

Course Logistics

Reference Reading:
Real-Time Embedded Systems: Chapter 3
Exploring BeagleBone. 2nd edition: Chapter 3, 5

Operating Systems

- **An operating system** provides an environment for execution of programs and services to programs and users
- One set of **operating system services** provides functions that are helpful to the user:
  - **User interface -** Almost all operating systems have a user interface (UI, i.e. command-line (Linux terminal, etc.), graphics user interface (GUI – Windows Desktop, Ubuntu, etc.), batch
  - **Program execution -** The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations -** A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation -** Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

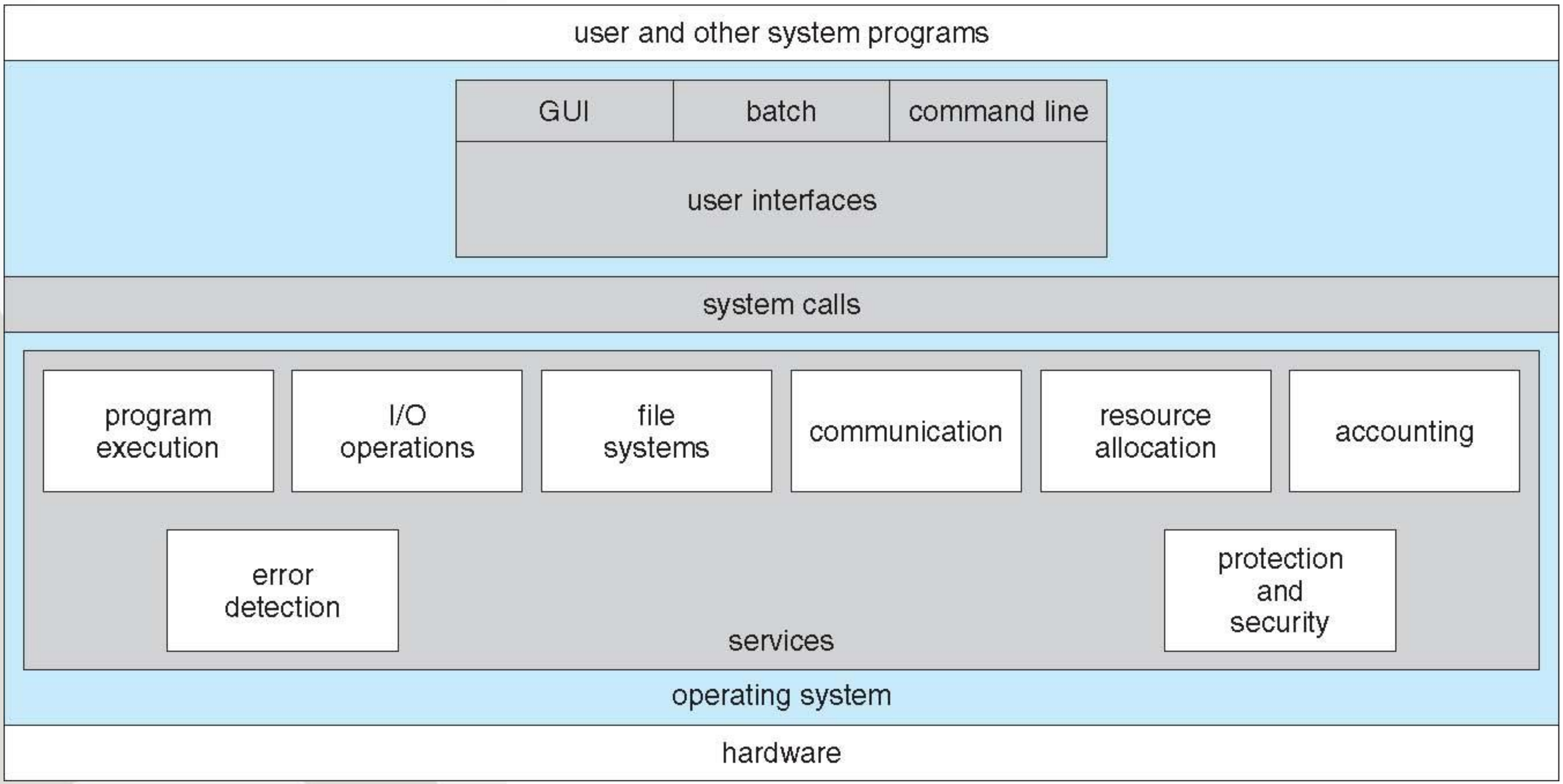
Operating System (Cont.)

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

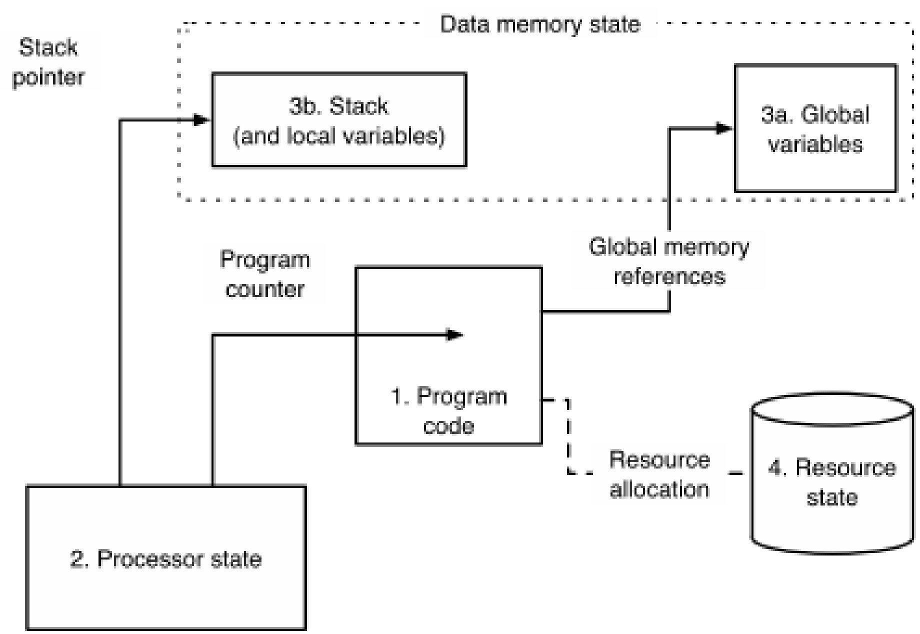
A View of Operating System Services



Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Terms ***job, tasks*** and ***process*** are used almost interchangeably
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

Process State Components



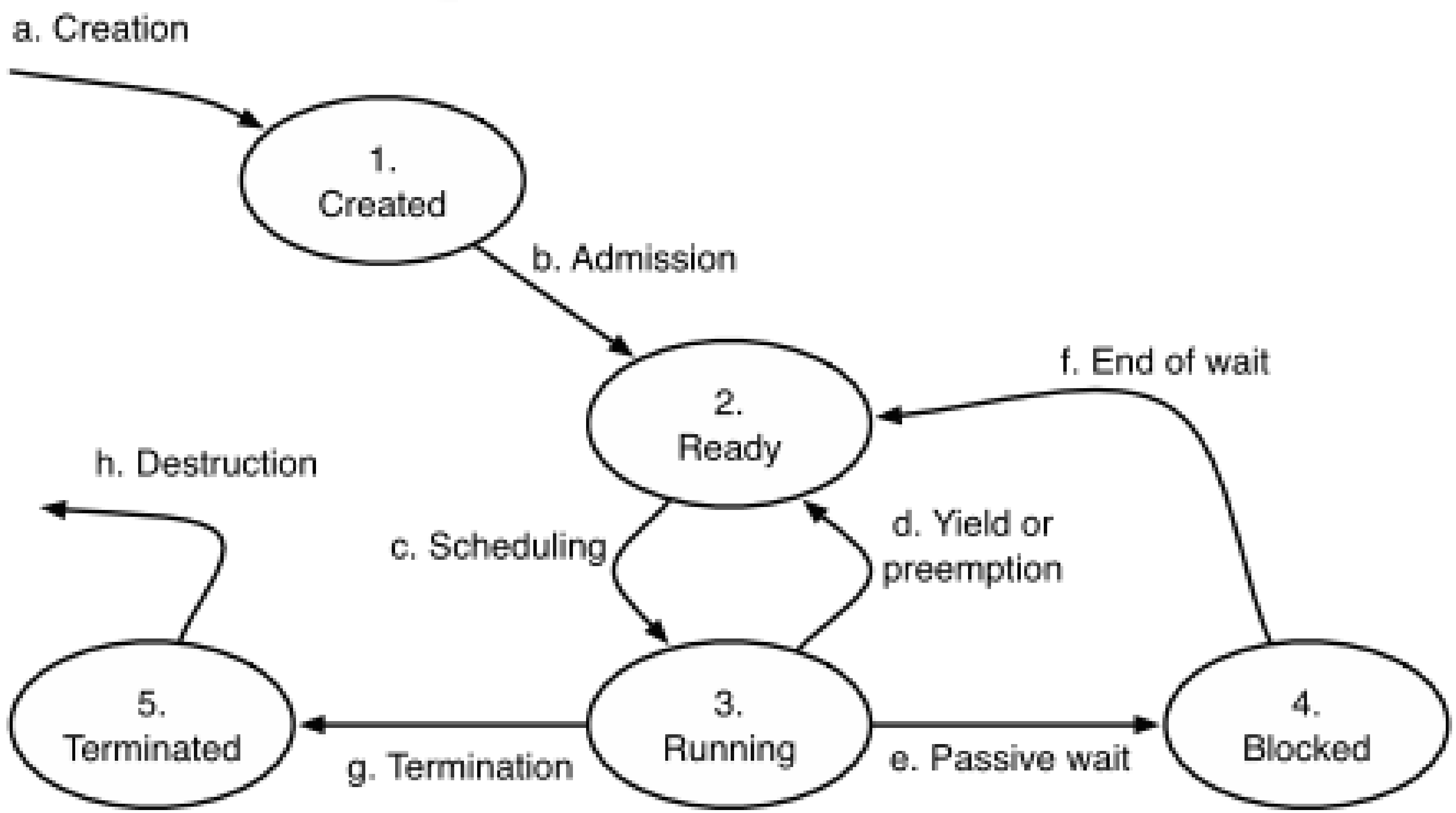
- **Process** – a program in execution; Process execution must progress in sequential fashion
- Main process state components:

1. **Program code**, also called text of the program;
2. **Processor state:** program counter, processor registers, status words, etc.
3. **Data memory** containing global variables, procedure call stack, local variables
4. **State of OS resources currently assigned to, and being used by the process**: open files, input-output devices, etc.

Process State

- As a process executes, it changes **state**
  - **created:** The process was just created
  - **ready:** The process is waiting to be assigned to a processor
  - **running:** The process is being executed
  - **blocked:** The process is waiting for some event to occur
  - **terminated:** The process has finished execution

Process State Diagram

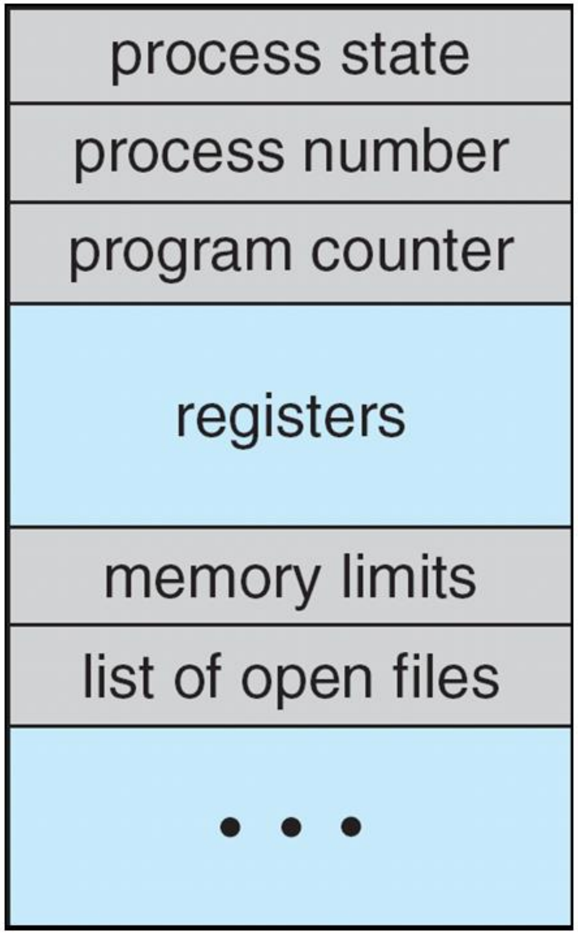


Process Control Block (PCB)

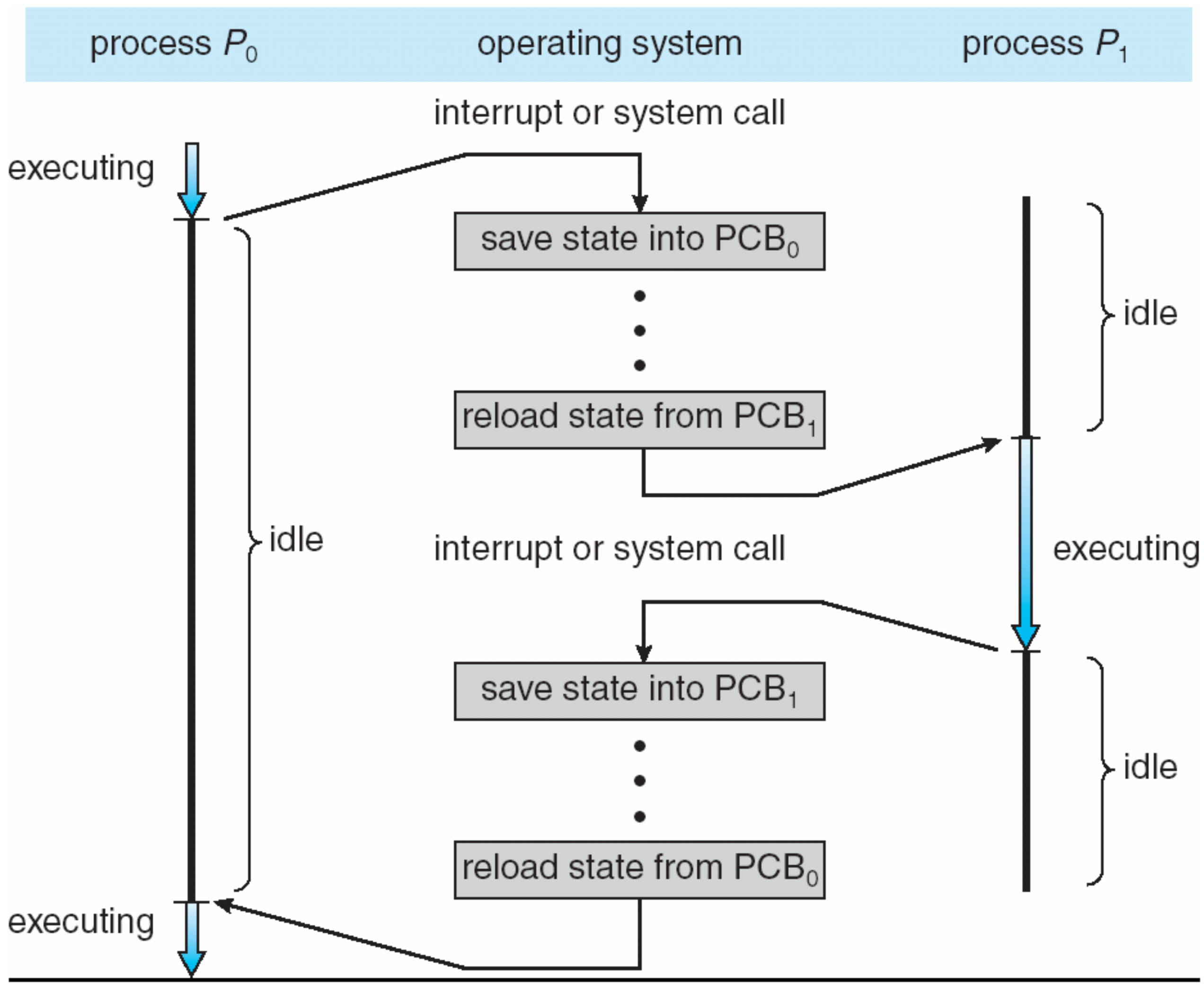
Each process is represented by a **process control block** (also called **task control block**)

Contains information associated with each process:

- **Process state** – running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information** – priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



Concurrency

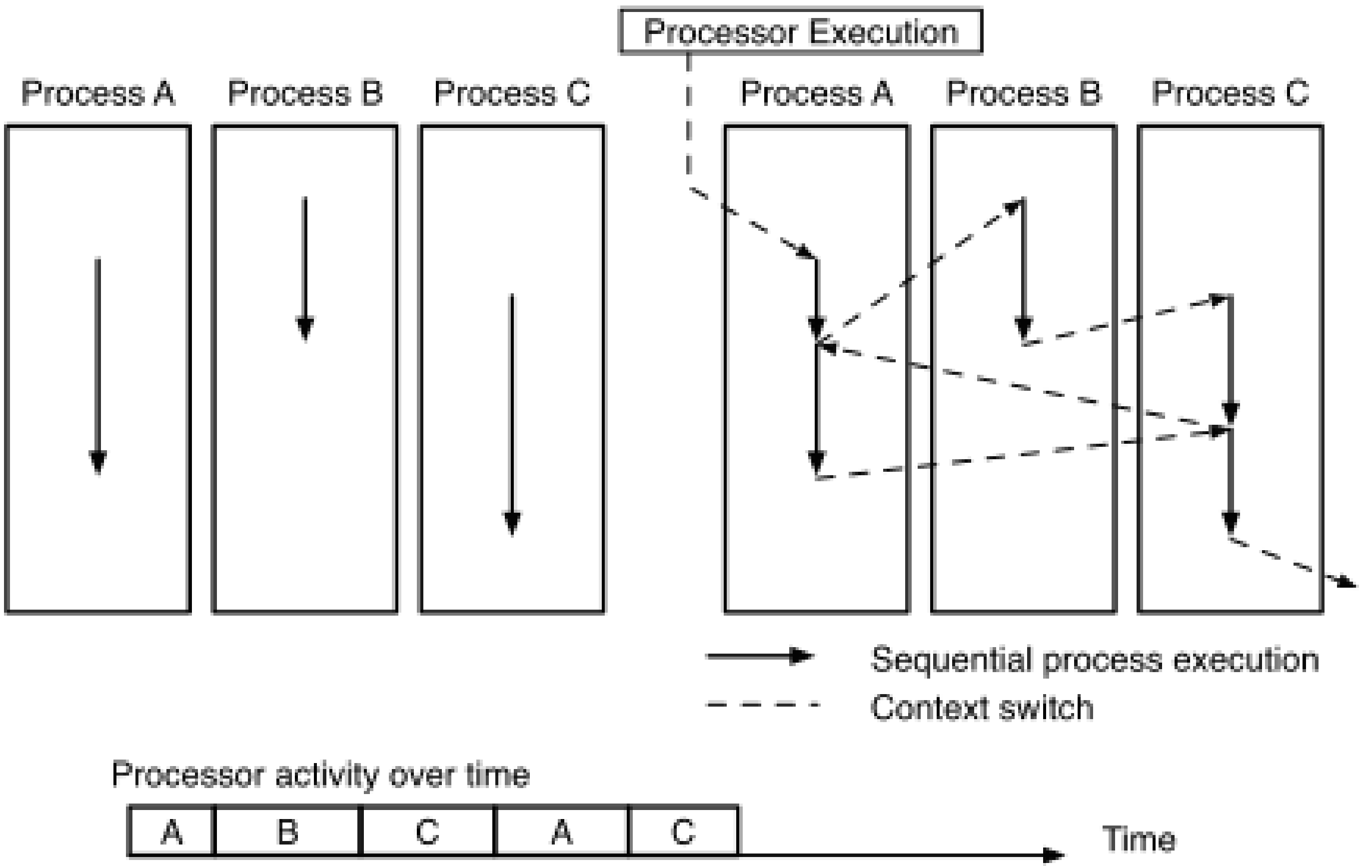
- In real life we act on a number of concurrent stimuli, say reading and at the same time listening to music.
- The instrumentation in a car collects and display continuous information about speed, fuel consumption etc, regulate the indoor temperature according to a set value, monitors the status of oil, brakes and whatever, controls the speed by means of the cruise control system.
- The ABS (Anti-Lock Braking System) prevents locking of a vehicle's wheels during braking.
- All together, most measurement and control systems are inherently concurrent.

True Concurrency vs. Pseudo-Concurrency

- Computer science defines concurrency as a property of systems where several processes are executing at the same time, and may or may not interact with each other.
- In a single processor, concurrent external activities must be ”mapped” as pseudo-concurrent sequential processes. The timing requirements is met when all the sequential processes can react within the given deadlines.
- True concurrency requires parallell processing in separate processors, either a multi-processor system or multi-CPUs
- However, in most cases several tasks can be executed **pseudo-concurrently** in a single processor.

Pseudo-Concurrency: Multiprogramming Concept

Executing three processes on a single-processor system

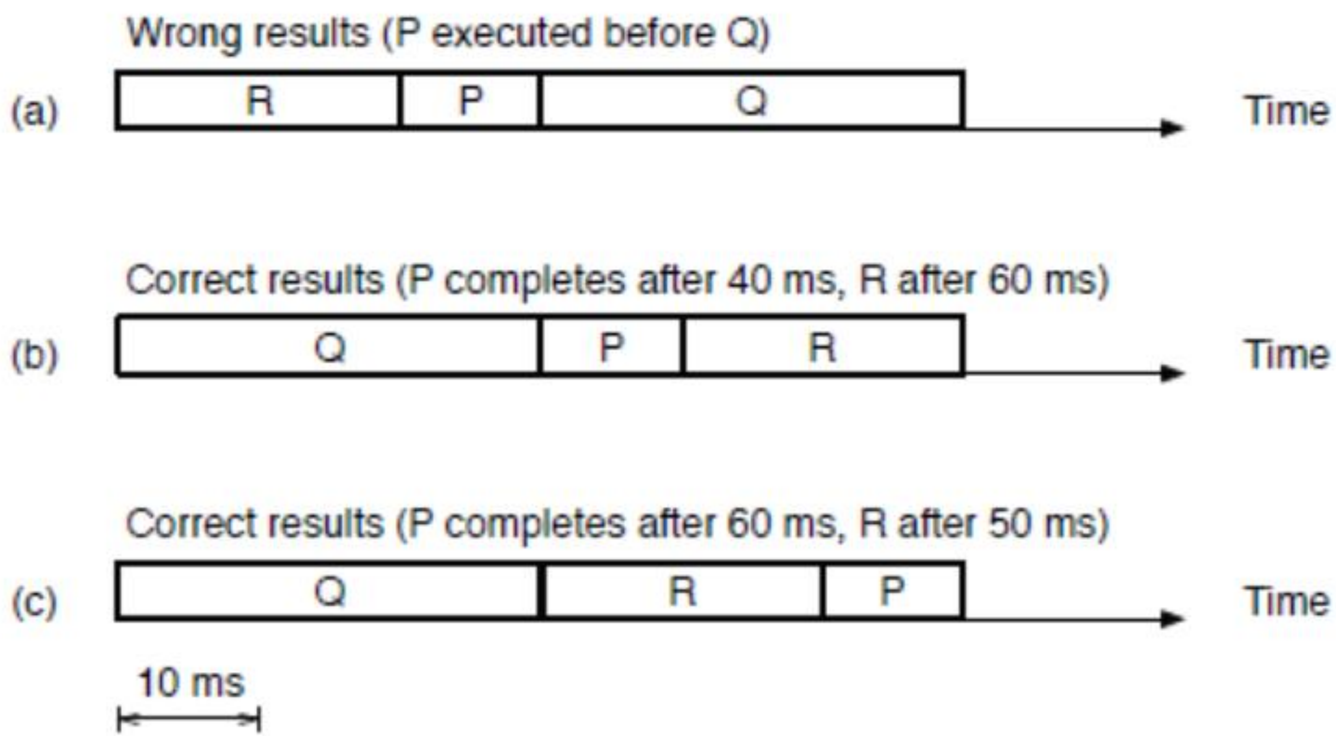


Concurrent Programming

- Concurrent programming notation expresses potential parallelism and dealing with **synchronization** and **communication** problems
- The responsibility of choosing which processes will be executed at any given time by the available processors, and for how long, falls on the operating system and, in particular, on an operating system component known as **scheduler**.
- If a set of processes must cooperate to solve a certain problem, not all possible choices will produce meaningful results.

Concurrent Programming

- Consider processes *P*, *Q* and *R*.
- *Q* produces some data used by *P*, so *P* cannot be executed before *Q*



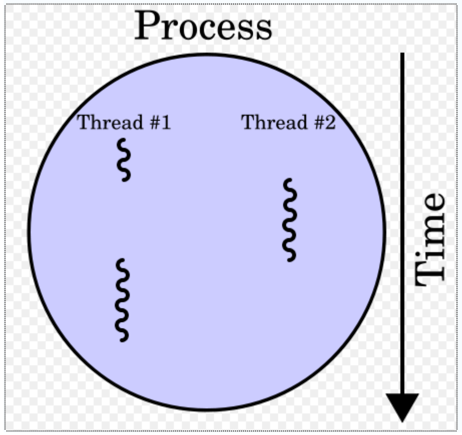
Execution of Real-Time Processes – How?

- A process can be executed, or resumed after a wait, by:
  - User/operator command
  - Data **interrupt**, for instance from the ADC
  - By a clock (timer), typical for a **periodic** execution
    - Timer interrupt
    - On a given date and time
  - By a **signal** or **message** from another process
- If several processes want to execute at the same time, a mechanism is required to select which process shall get the CPU

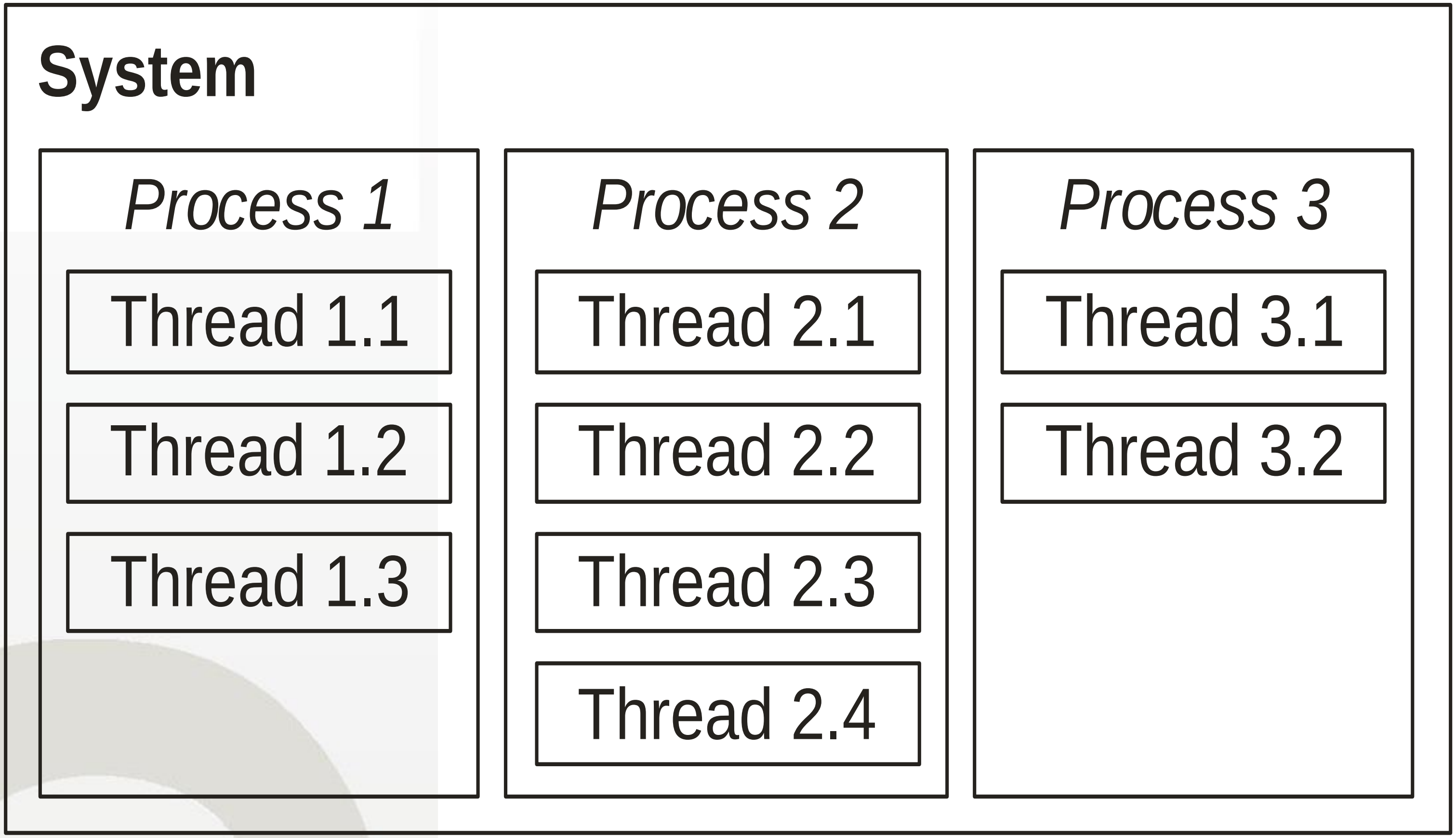
Process vs. Threads

A process (synonymously called "**task**") is
- An abstraction of a running program
- The logical unit of work schedulable by the OS

A thread is a lightweight process within a regular process. It has access to the same memory space, and the context switching from one thread to another will be shorter than for process to process

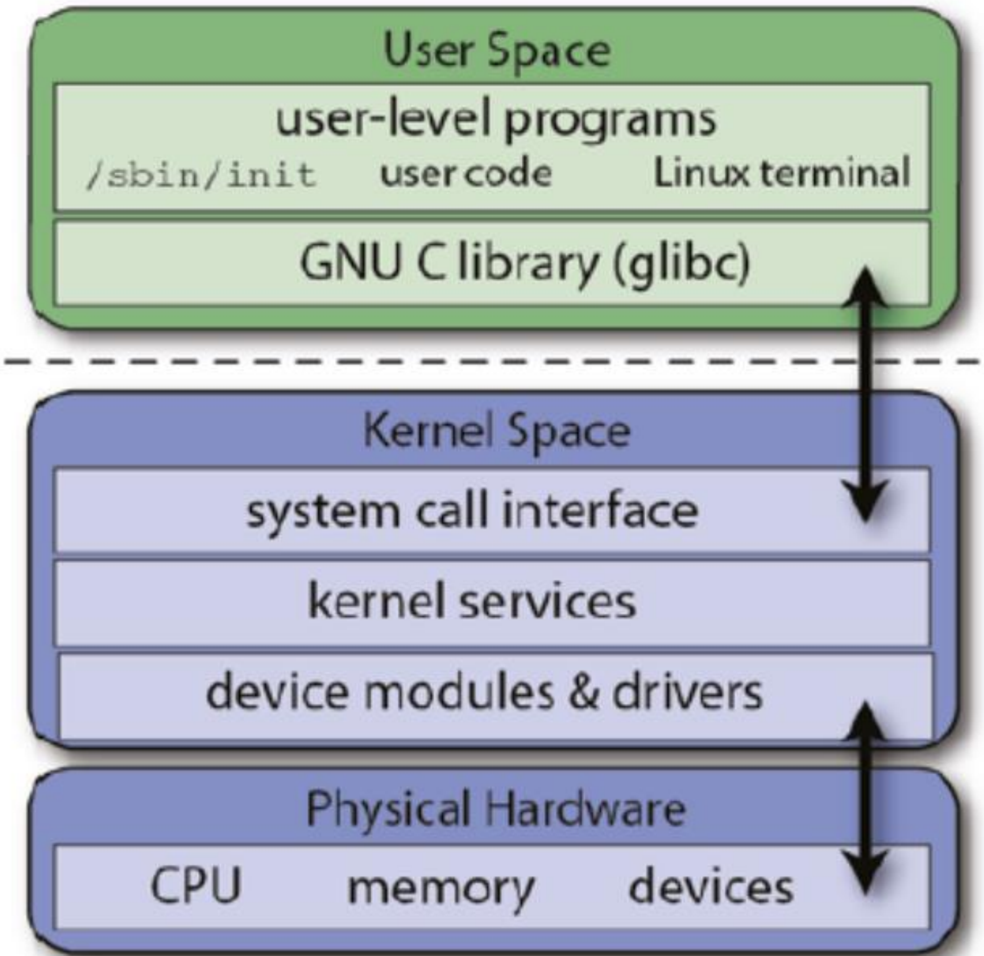


Processes, and Multiple Threads



Kernel Space and User Space

The Linux user space and kernel space architectures



A hard boundary between user and kernel spaces prevents user applications from accessing memory and resources required by the Linux kernel:

- this helps preventing the Linux kernel from crashing because of badly written user code;
- It prevents applications that belong to one user from interfering with applications and resources that belong to another user,
- it also provides a degree of security.

Kernel Tasks

- **Kernel -** the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

1. Process scheduling:
   - Linux is a preemptive multitasking operating system, Multitasking means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s).
2. Memory management:
3. Provision of a file system
4. Creation and termination of processes:
5. Access to devices
6. Networking:
7. Provision of a system call API

System Calls

- **Programming interface to the services provided by the OS**
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls (Cont.)

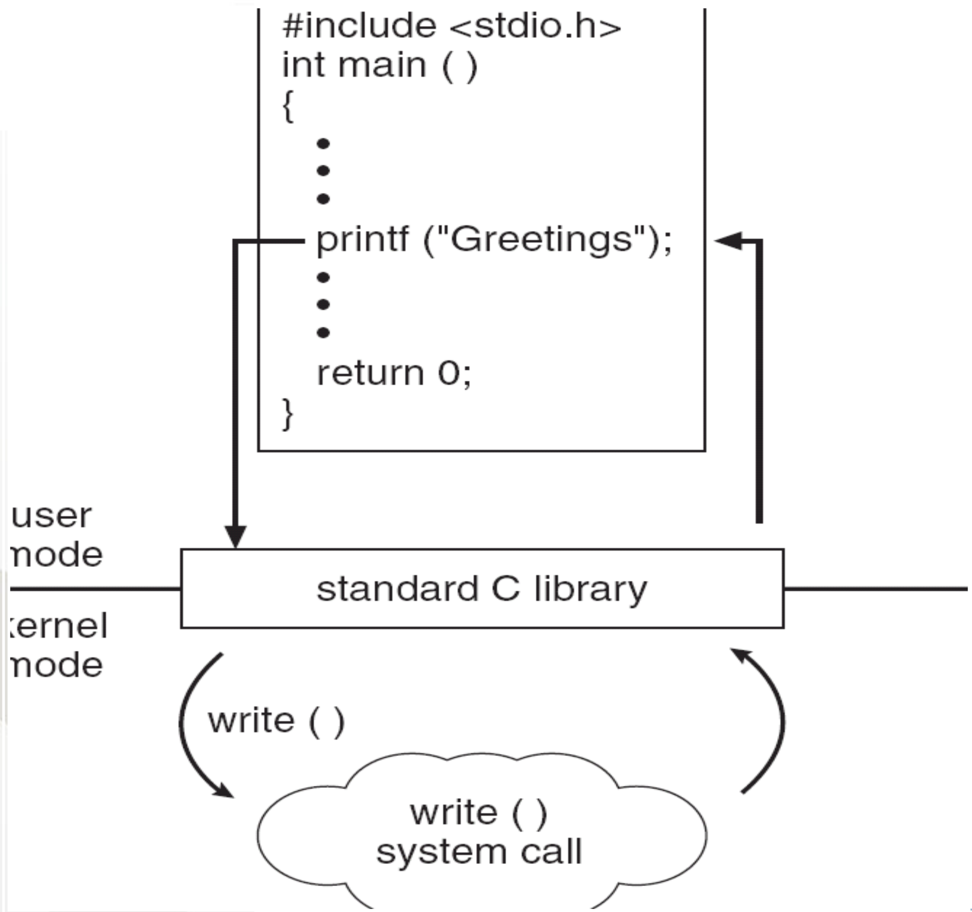
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Standard C Library Example

- C program invoking printf() library call, which calls write() system call

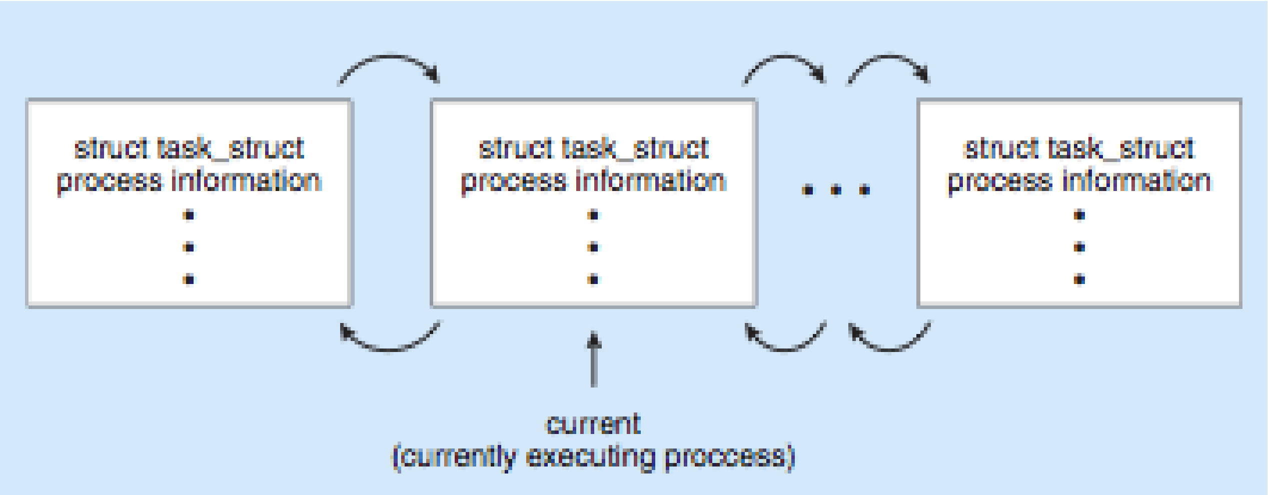


Process Creation

- Generally, process is identified and managed via **a process identifier (pid)**
- **Parent** process creates **children** processes, which, in turn, create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

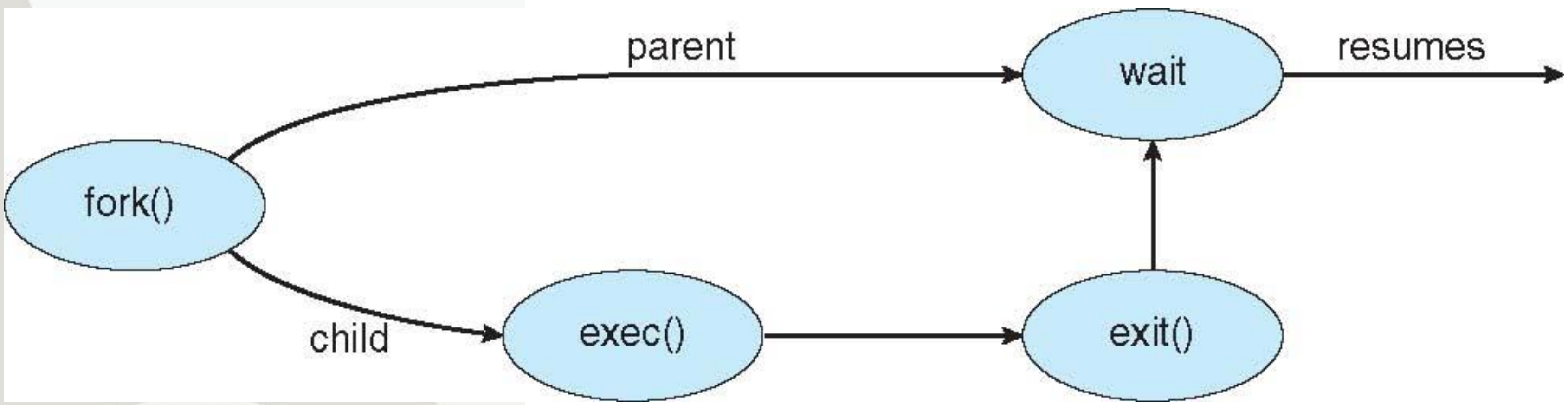
Process Representation in Linux

Represented by the C structure **task_struct**
pid t_pid /* process identifier */
long state /* state of the process */
struct sched_entity se /* scheduling information */
struct task_struct *parent /* this process's parent */
struct list_head children /* this process's children */
struct files_struct *files /* list of open files */
struct mm_struct *mm /* address space of this process */



Linux (UNIX) Process Creation

- Linux examples
  - **fork** system call creates new process
  - **exec** system call is used after a **fork** to replace the process' memory space with a new program



The Linux (UNIX) system call Fork()

- **fork()** takes no arguments and returns a process ID.
- The purpose of **fork()** is to create a ***new*** process, which becomes the ***child*** process of the caller.
- After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.
- Returned value of **fork()**:
  - If **fork()** returns a negative value, the creation of a child process was unsuccessful.
  - **fork()** returns a zero to the newly created child process.
  - **fork()** returns a positive value, the ***process ID*** of the child process, to the parent
- UNIX will give an exact copy of the parent's address space and give to the child

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          return 1;
   }
   else if (pid == 0) { /* child process */
          execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
          /* parent will wait for the child */
          wait (NULL);
          printf ("Child Complete");
   }
   return 0;
}
```

Fork() example 2

The programs runs two processes: a parent and a child

Both of them run the same loop and print some messages

```
#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  200

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}
void  ChildProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("   This line is from child, value = %d\n", i);
     printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent, value = %d\n", i);
     printf("*** Parent is done ***\n");
}
```

Any Questions?