Introduction to threads

Juan Felipe Medina Lee, Ph.D.

Computer Science and Engineering
University of Puerto Rico Mayagüez Campus

Overview

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries

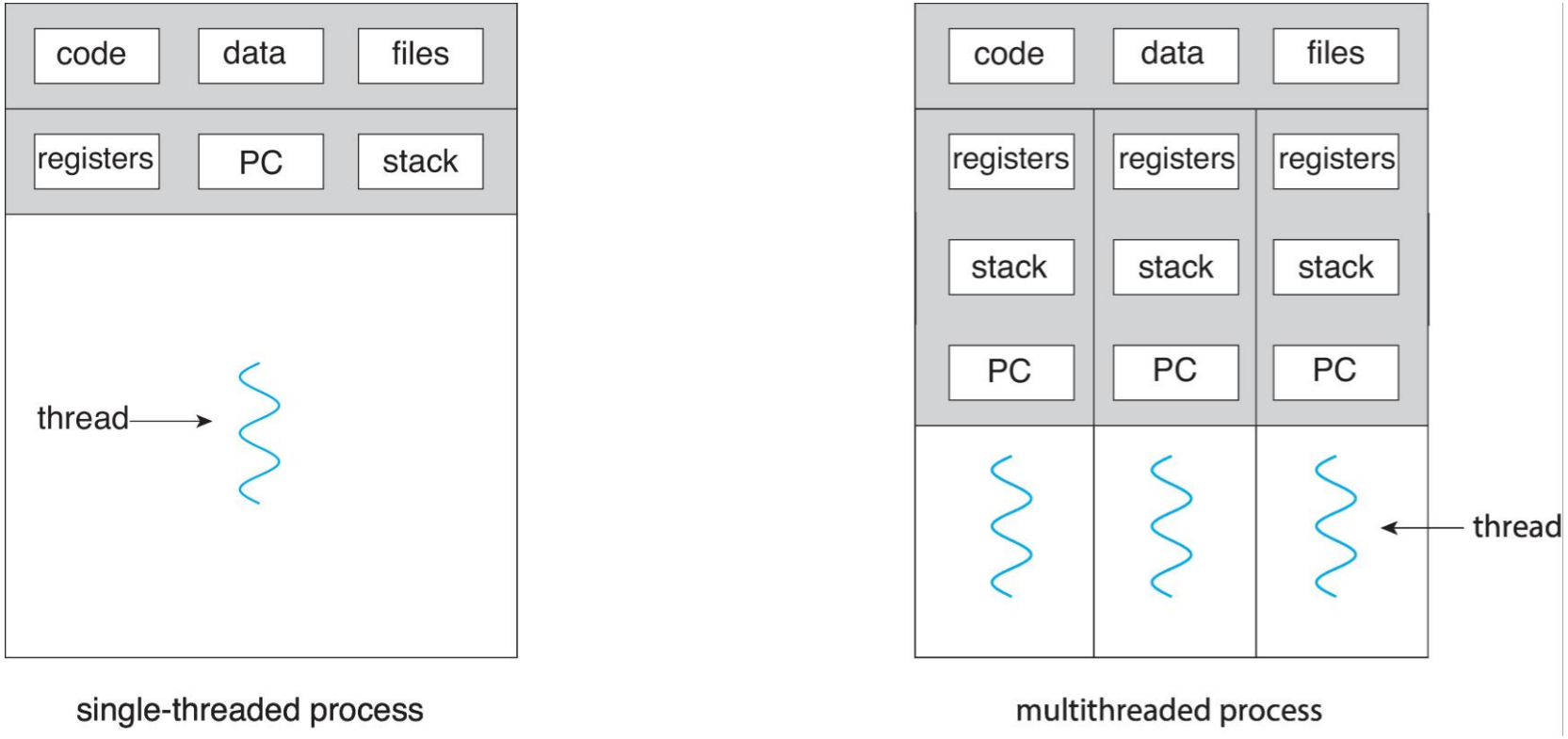
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks in the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

What is a thread?

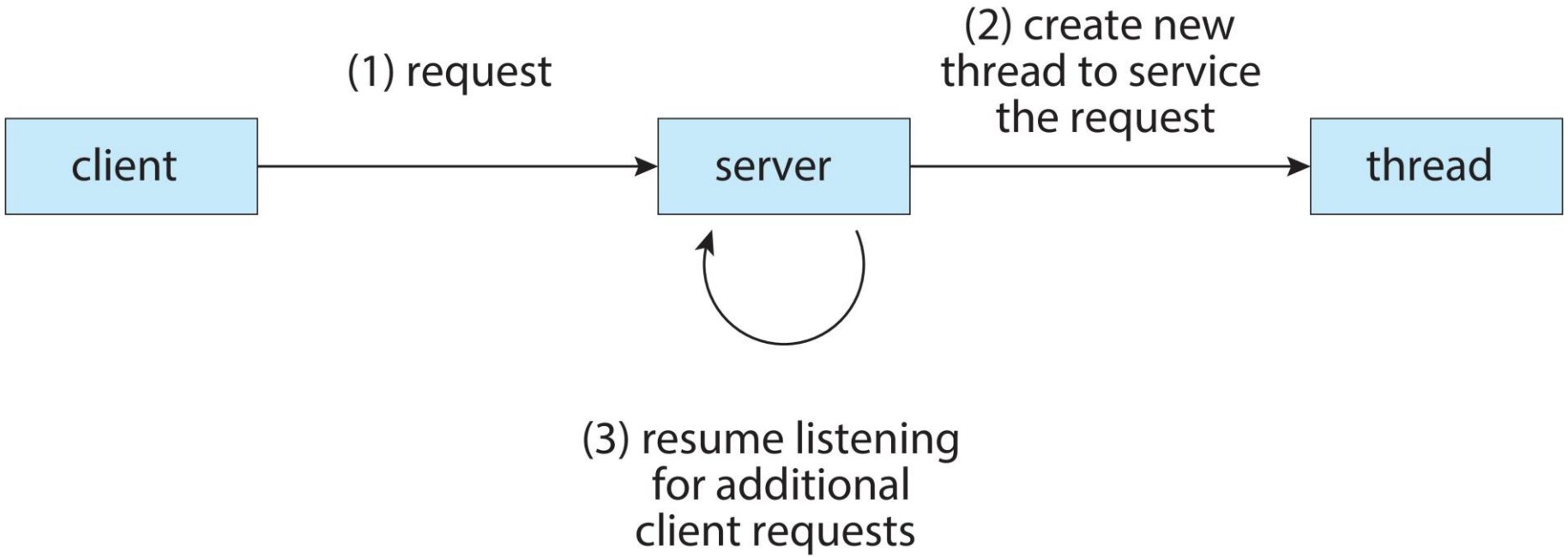
- A **thread** is a basic unit of CPU utilization within a process. In computing, it represents a single sequence of executed instructions that can run concurrently with other threads within the same process.
- Each thread shares the same resources with other threads in the process
  - Memory
  - File descriptors
  - Global variables
- Nevertheless, they operate independently in terms of execution flow.

Single and multithreaded processes



single-threaded process

multithreaded process



Multithreaded server architecture

Benefits

Responsiveness – This may allow continued execution if part of the process is blocked, which is especially important for user interfaces.

Resource Sharing – threads share resources of process, which is easier than shared memory or message passing.

Economy – cheaper than process creation, thread switching lower overhead than context switching.

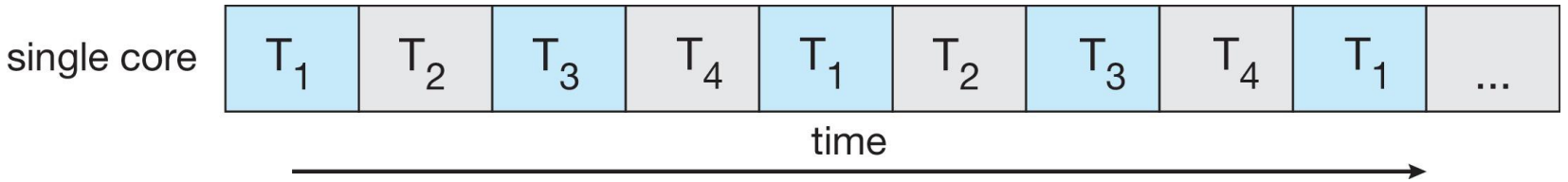
Scalability – process can take advantage of multicore architectures.

Multicore programming

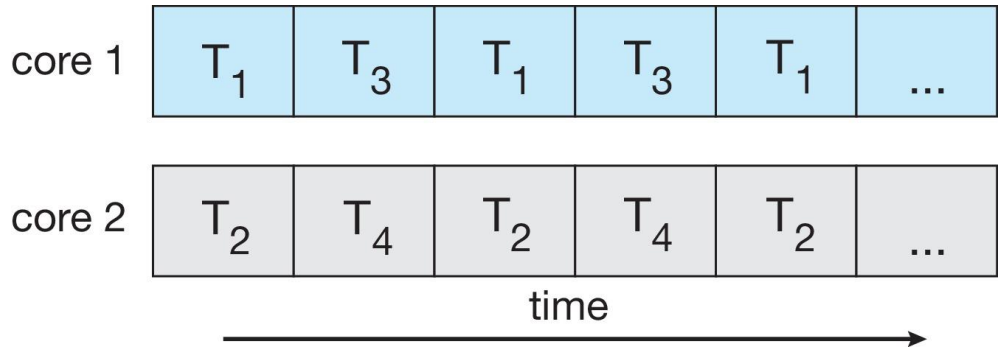
- Multicore or multiprocessor systems put pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than one task making progress
  - Single processor/core, scheduler providing concurrency

Concurrency vs parallelism

- Concurrent execution on a single-core system:



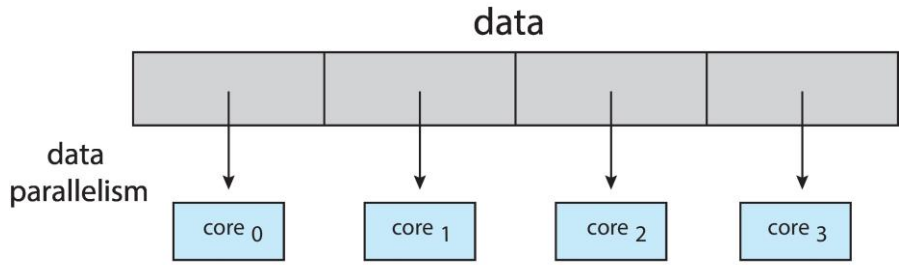
- Parallelism on a multi-core system:



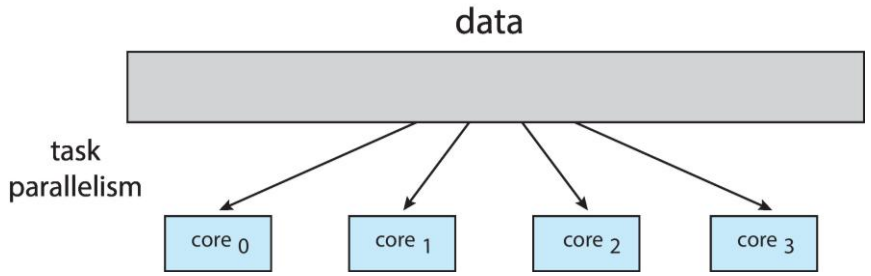
Concurrency vs parallelism

Types of parallelism:

- Data parallelism – distributes subsets of the same data across multiple cores, same operation on each



- Task parallelism – distributing threads across cores, each thread performing unique operation



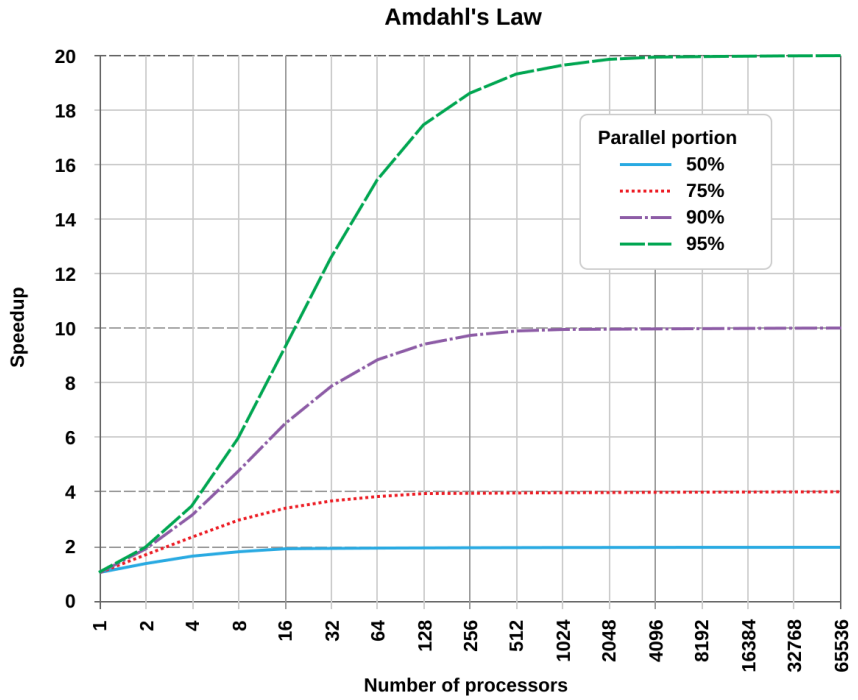
Ahmdal’s Law

Identifies performance gains from adding additional cores to an application that has both serial and parallel components

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

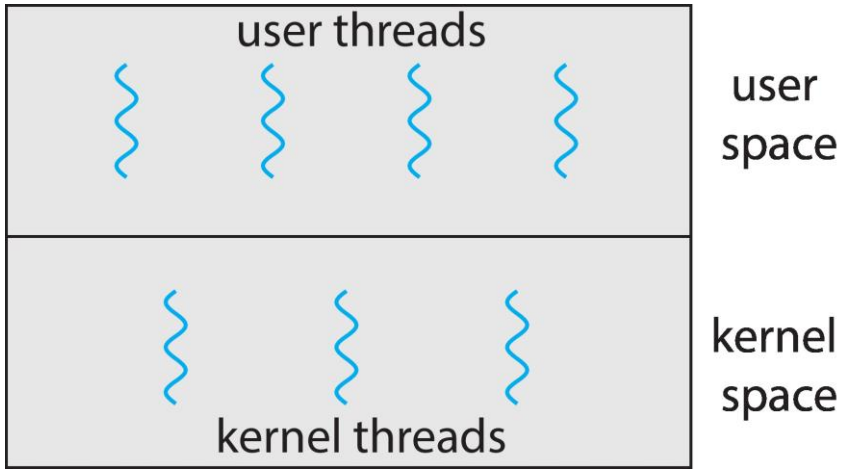
Where:

- **S(N)** is the speedup with **N** processors.
- **P** is the proportion of the program that can be parallelized.
- **(1 - P)** is the proportion of the program that is sequential and cannot be parallelized.
- **N** is the number of processors or cores.



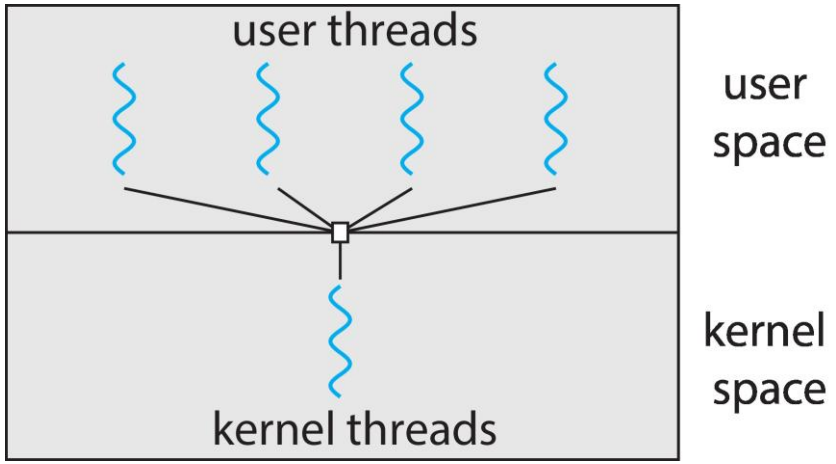
User threads and kernel threads

- User threads - management is done by user-level threads library
- Kernel threads - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android



Multithreading models: Many to one

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



Multithreading models: Many to one

Each user-level thread maps to kernel thread

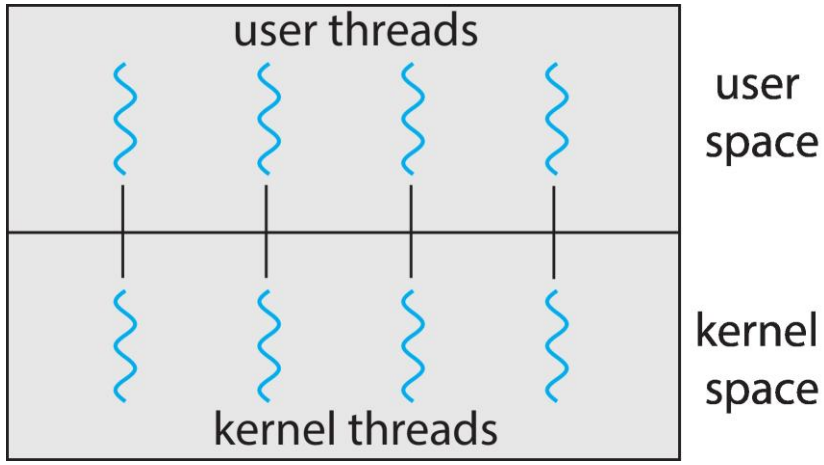
Creating a user-level thread creates a kernel thread

More concurrency than many-to-one

Number of threads per process sometimes restricted due to overhead

Examples

- Windows
- Linux



Posix threads

PTHREADS

Pthreads

- In GNU/Linux, it provides threads at kernel level
- A POSIX standard (IEEE 1003.1c) **API** for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies the behavior of the thread library. Implementation is up to the development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

Pthreads functions (1)

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t * attr, void
* (*start_routine)(void *), void * arg);
```

- **`pthread_create`** creates a new thread of control that executes concurrently with the calling thread.
- The new thread applies the function **`start_routine`**, passing `arg` as the first argument.
- The new thread terminates explicitly by calling `pthread_exit` or implicitly by returning from the **`start_routine`** function.
  - Returning a value is equivalent to calling **`pthread_exit`** with the result value as exit code.
- The `attr` argument can also be `NULL`, in which case the default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

Pthreads functions (2)

```
void* function(void* param)
```

- The function associated with a thread **must** return a void pointer and receive a void pointer

Pthreads functions (3)

- The pthread_attr_t is a data type in the POSIX threads (pthreads) library that is used to specify attributes when creating new threads. It provides a way to define various properties of a thread, such as:
  - its stack size
  - scheduling policy
  - the thread should be joinable or detached.
- By using the pthread_attr_t structure, you can customize thread behavior beyond the default settings when calling pthread_create().

```
pthread_attr_t attr;

pthread_attr_init(&attr);
```

Pthreads functions (4)

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

`pthread_join` suspends the execution of the calling thread until the thread identified by *th* terminates, either by calling `pthread_exit` or by being canceled.

Thank you

Juan F. Medina

Juan.medina26@upr.edu

Mutual exclusion (mutexes)

Juan Felipe Medina Lee, Ph.D.

Computer Science and Engineering
University of Puerto Rico Mayagüez Campus

Overview

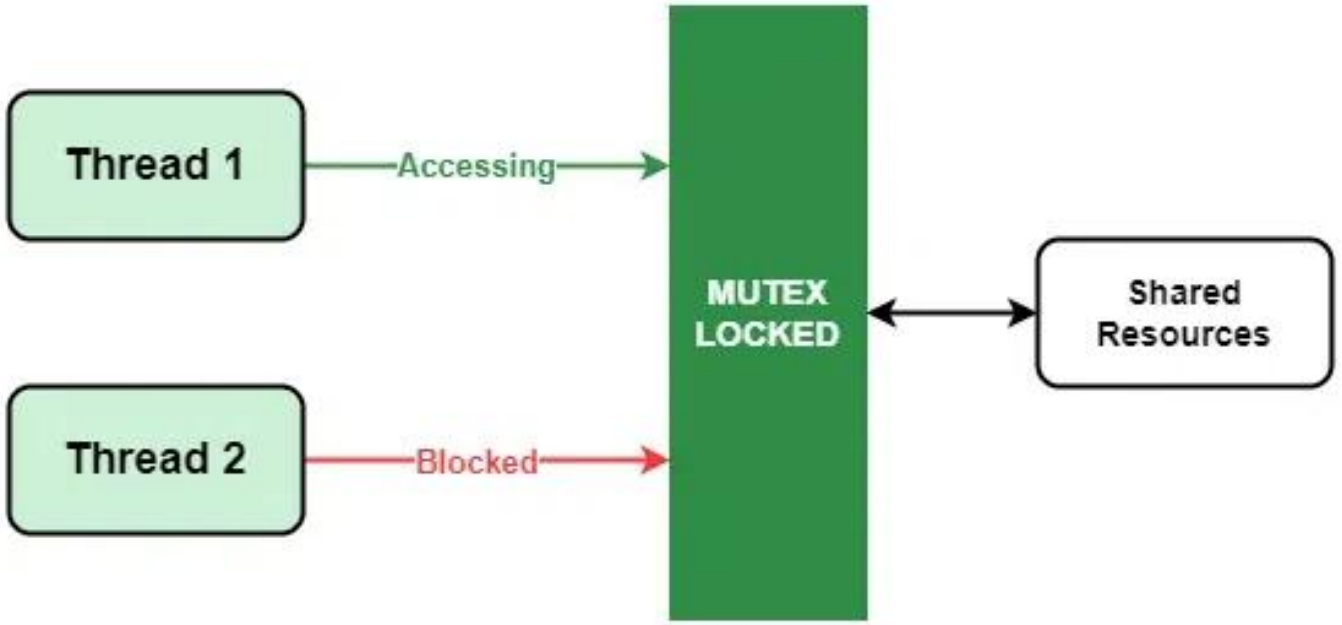
- Overview
- usage
- Implementation in pthread

What is a mutex?

- A mutex (short for *mutual exclusion*) is a synchronization primitive that prevents multiple threads from accessing a shared resource or critical section of code simultaneously.

- The purpose of a mutex is to ensure that only one thread can access the shared resource at a time, thereby
  - avoiding *race conditions*
  - ensuring *data integrity*.

How does a mutex works?



- Using a mutex, each thread can lock the access to shared resources.
- Other threads have to wait for the thread to unlock the access.

Mutex in Posix (1)

```
pthread_mutex_init (mutex,attr)
pthread_mutex_destroy (mutex)
pthread_mutexattr_init (attr)
pthread_mutexattr_destroy (attr)
```

- Mutex variables are type **`pthread_mutex_t`**
- Attr variables are type **`pthread_mutexattr_t`**
  - It allows to modify properties like recursiveness or robustness
- Mutexes always are initialized **unlocked.**

Mutexes in Posix (2)

```
pthread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)
```

- **`pthread_mutex_lock:`** initializes a mutually excluded zone.
  - If another thread has locked the mutex, the thread becomes locked.
- **`pthread_mutex_unlock (mutex):`** Find if getting into the mutually excluded zone is possible.
  - If the mutex is locked, the function returns an error, but it does not get locked.
  - It's helpful to prevent **deadlocks** and **priority inversion.**

Conditional variables

- **Conditional Variable**: A condition variable is a synchronization primitive that enables threads to wait for certain conditions to be met before continuing. It allows one or more threads to "sleep" until another thread signals that the condition has been met.

- I’m willingly giving you the lock control so you can finish your work while I sleep.

Conditional variables in posix

- How `pthread_cond_wait()` Works:
  - A thread calls `pthread_cond_wait()` when it needs to wait for a certain condition.
  - Before calling `pthread_cond_wait()`, the thread must lock a mutex.
  - `pthread_cond_wait()` does two things:
    - It **unlocks the mutex** so that **other threads can acquire the lock** and change the shared data the thread is waiting for.
    - It puts **the thread in a blocked (waiting) state** until it is signaled by another thread that the condition has changed.

Example conditional variables

```
pthread_cond_t cond_var;
pthread_mutex_t mutex;
```

Global variables

THREAD A:

```
///
pthread_mutex_lock(&mutex);
///do some work
…
///willingly go to sleep so others can work
pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
```

THREAD B:

```
///
pthread_mutex_lock(mutex);
///do some work
…
///notify other threads that the work is done
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Thank you

Juan F. Medina

Juan.medina26@upr.edu

CPU Scheduling

Juan Felipe Medina Lee, Ph.D.

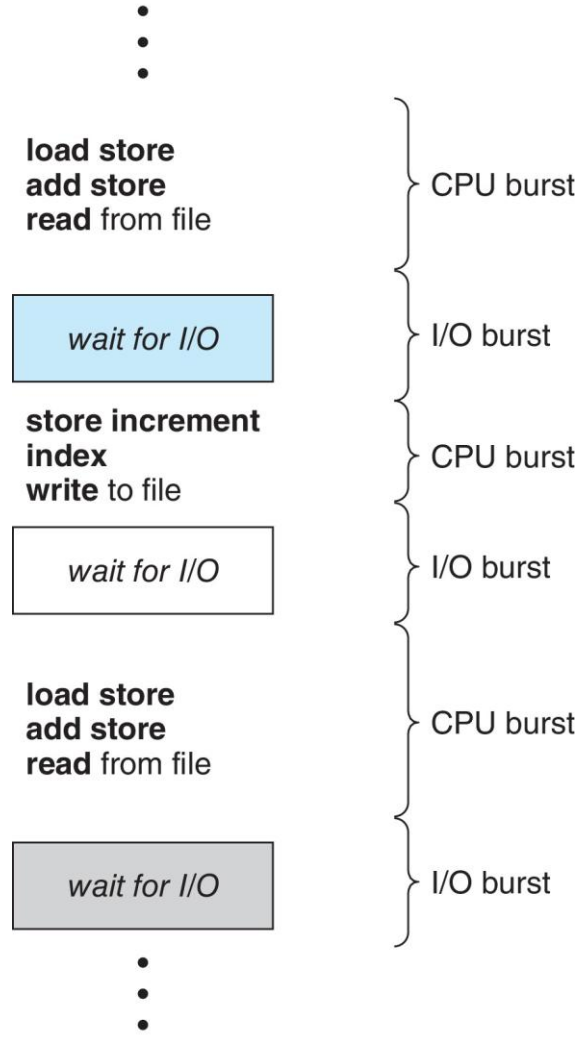
Computer Science and Engineering
University of Puerto Rico
Mayagüez Campus

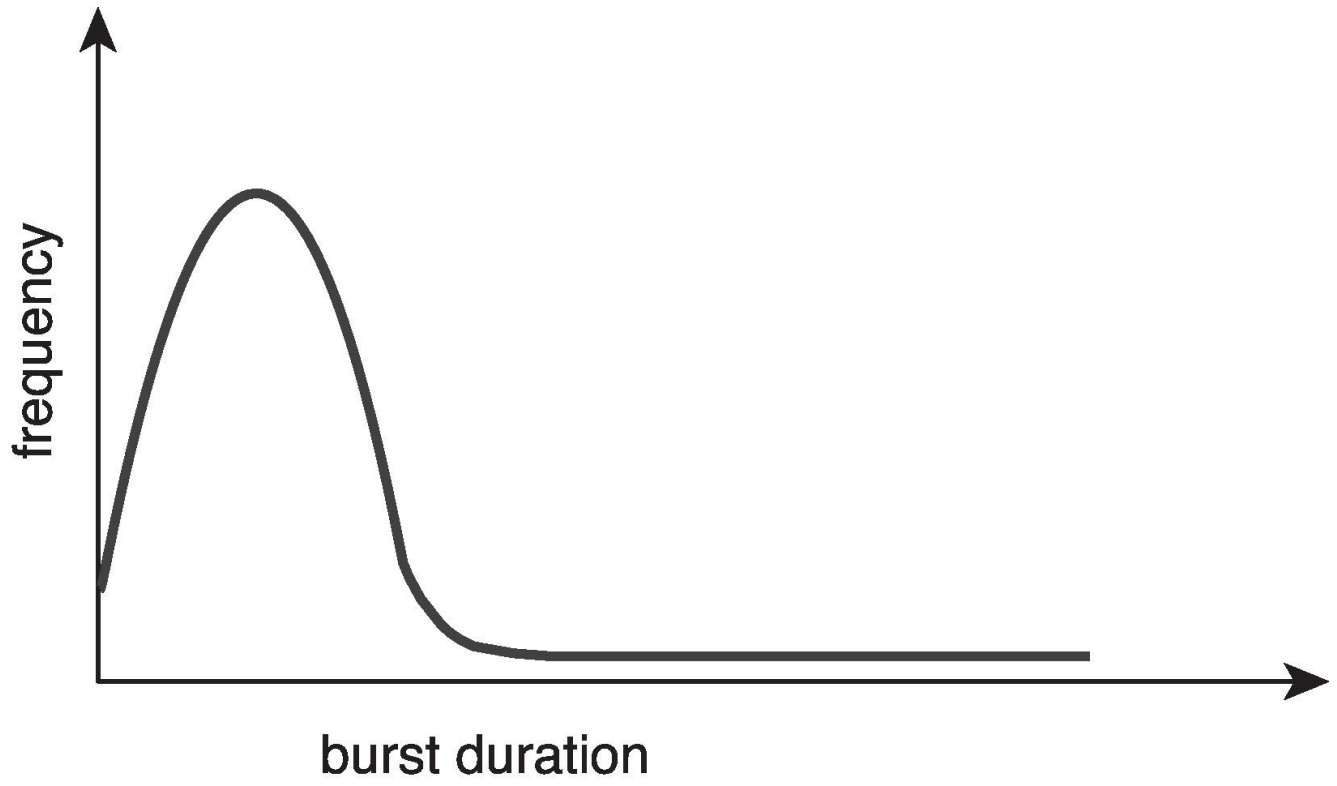
Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU Burst – I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



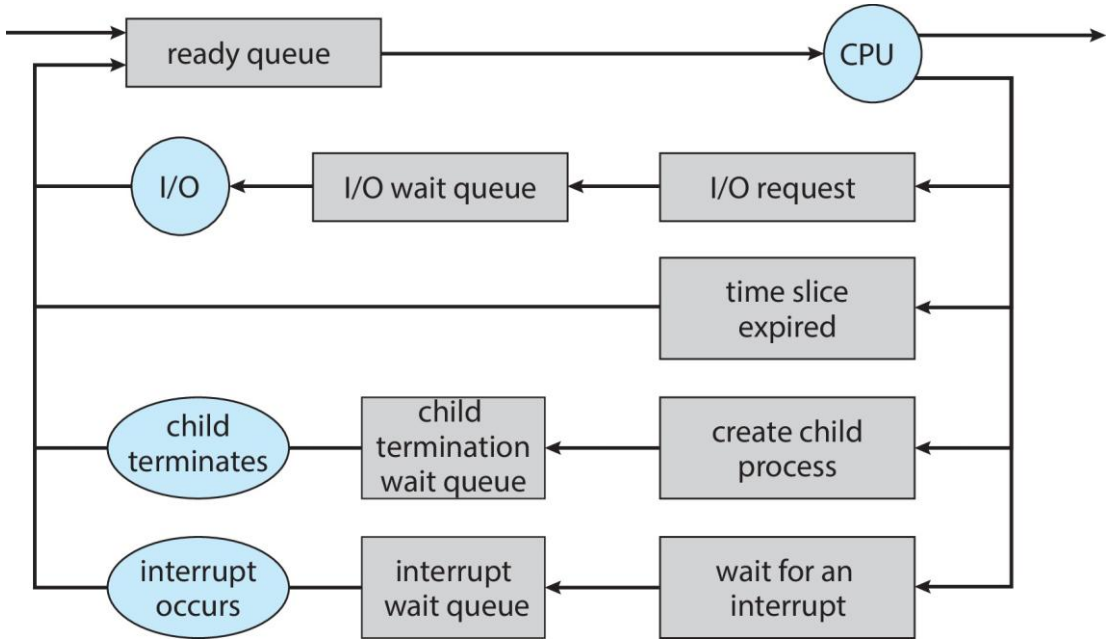


Histogram of CPU-burst Times

CPU Scheduler

- The **CPU scheduler** selects from among the processes in the ready queue and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- For situations 1 and 4, there is no choice regarding scheduling for that process. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.

Representation of Process Scheduling



Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **non-preemptive**.
- Otherwise, it is **preemptive**.
- Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems, including Windows, MacOS, Linux, and UNIX, use preemptive scheduling algorithms.

Preemptive Scheduling and Race Conditions

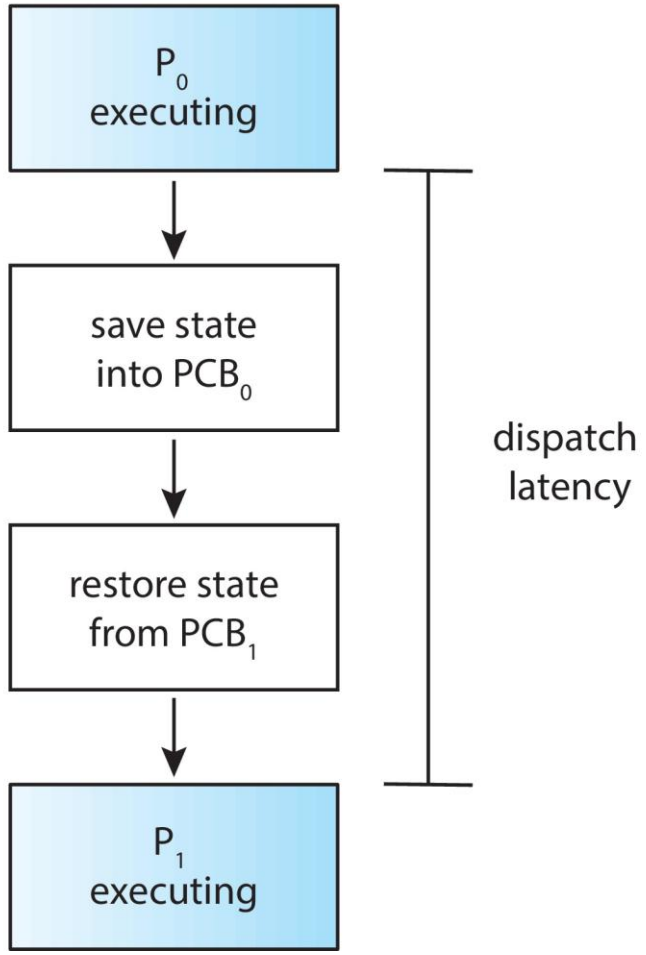
- Preemptive scheduling can result in race conditions when data are shared among several processes.

> **Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.**

- Commands of interest in Linux:
  - `vmstat 1 10`
  - `cat /proc/2166/status`

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – the amount of time to execute a particular process (waiting time + CPU time + I/O time)
- **Waiting time** – the amount of time a process has been waiting in the ready queue
- **Response time** – the time it takes from when a request was submitted until the first response is produced.

First- Come, First-Served (FCFS) Scheduling (AKA FIFO)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than the previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

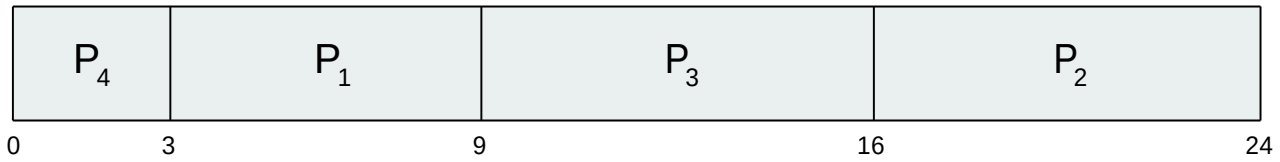
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its **next CPU burst**
  - Use these lengths to schedule the process in the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

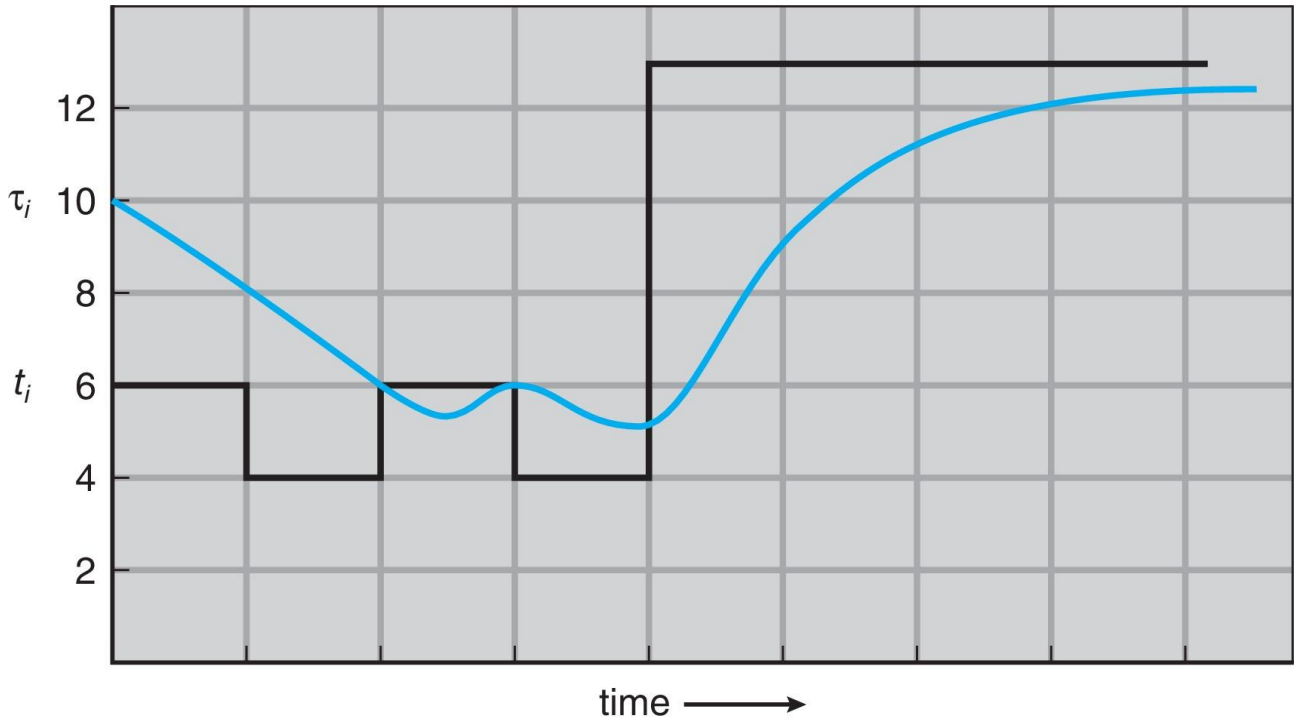
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick the process with the shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

- Commonly, α set to ½



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Prediction of the Length of the Next CPU Burst

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts

Shortest Remaining Time First Scheduling

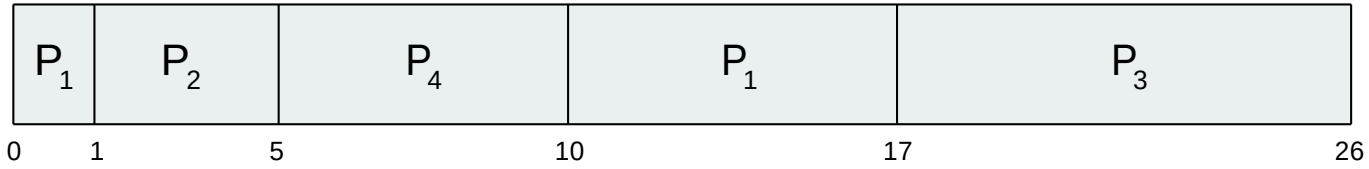
- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- **Is SRT more “optimal” than SJF in terms of the minimum average waiting time for a given set of processes?**

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process i	*Arrival* Time T	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

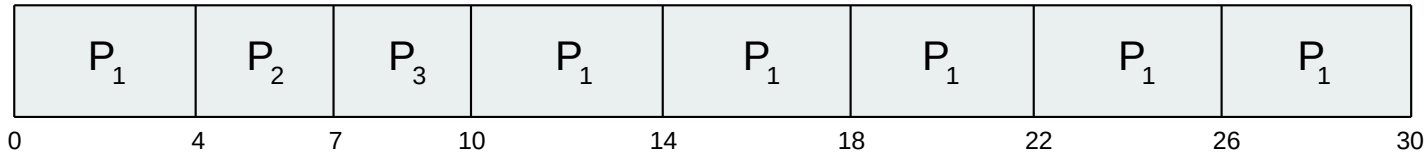
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO (FCFS)
  - q small $\Rightarrow$ RR
- Note that q must be large with respect to context switch, otherwise overhead is too high

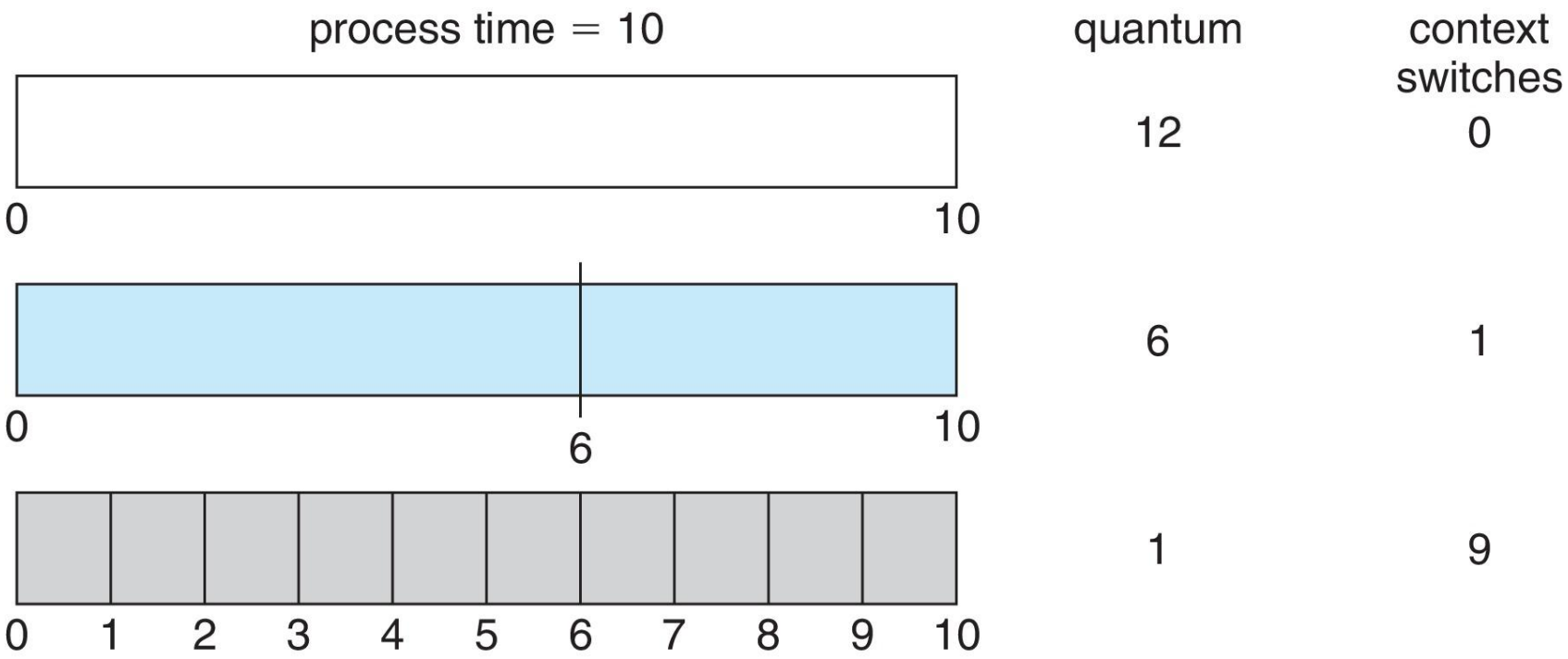
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

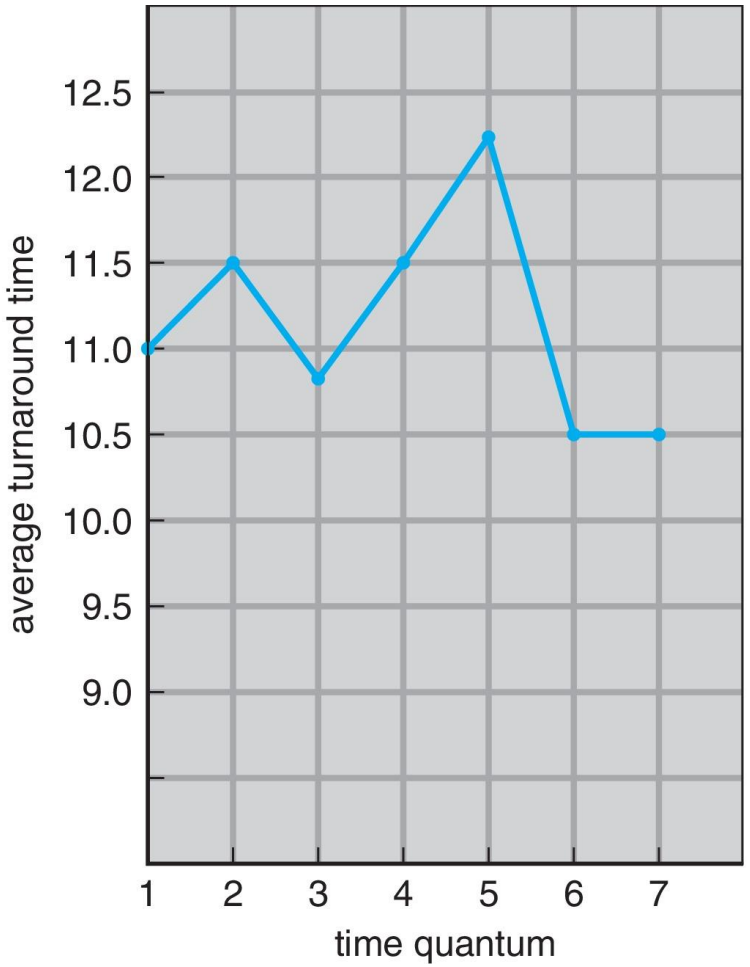


- Typically, **higher average turnaround** than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds



The performance of the RR algorithm depends heavily on the size of the time quantum.

Time Quantum and Context Switch Time



process	time
P_1	6
P_2	3
P_3	1
P_4	7

- 80% of CPU bursts should be shorter than TQ

Turnaround Time Varies With The Time Quantum

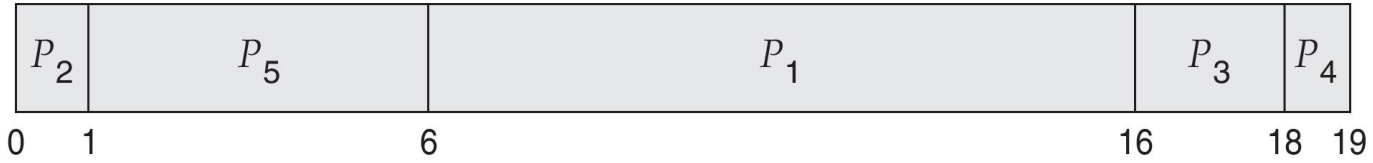
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
    - Preemptive
    - Nonpreemptive
- SJF is a priority scheduling where priority is the inverse of the predicted next CPU burst time
- Problem ≡ **Starvation** – low-priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process.

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



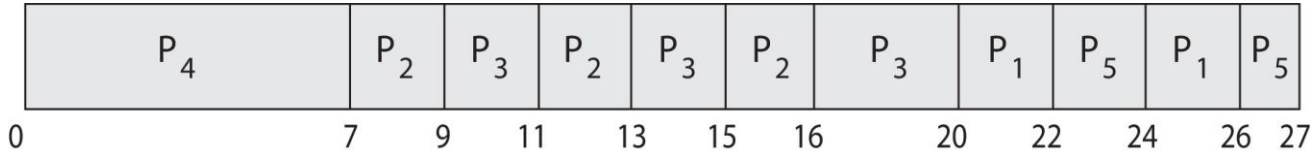
- Average waiting time = 8.2

Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Gantt Chart with time quantum = 2

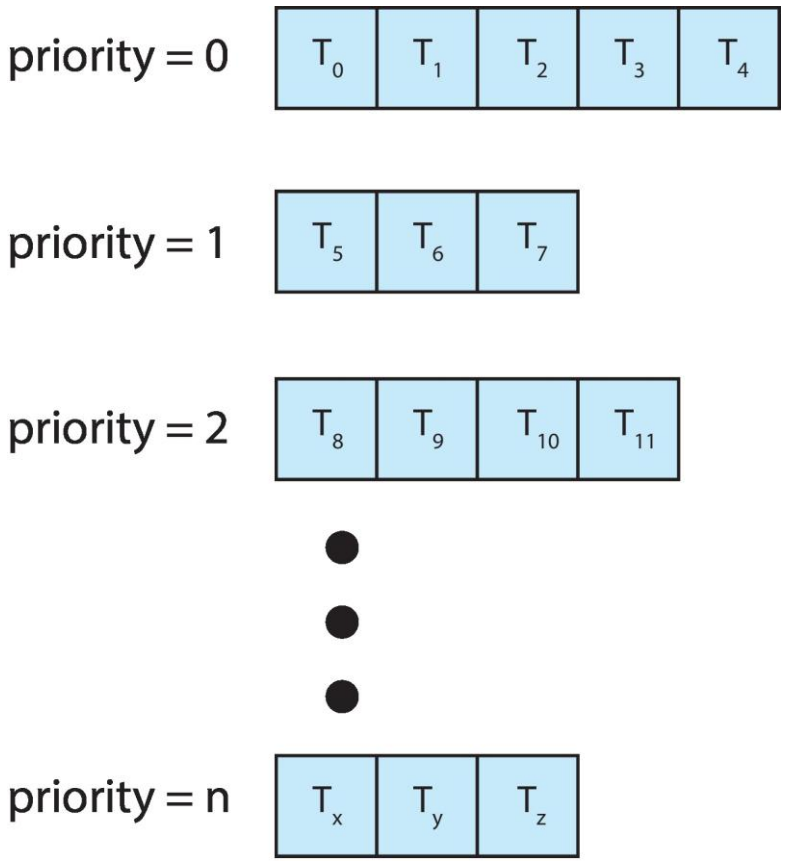


Multilevel Queue

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

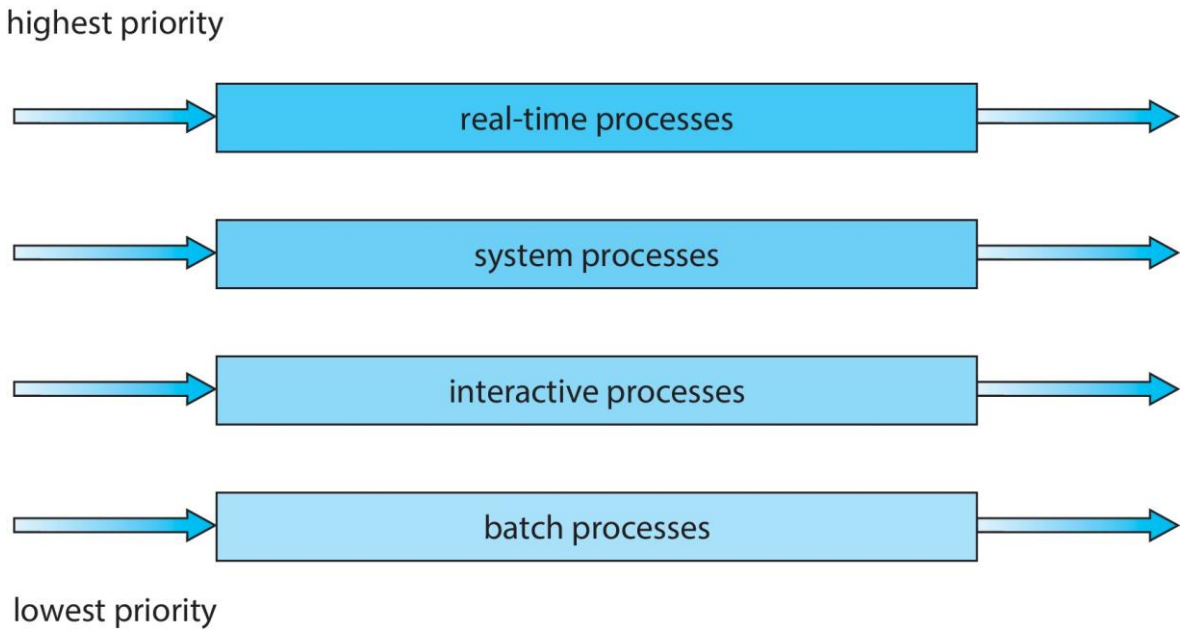
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

Prioritization based upon process type



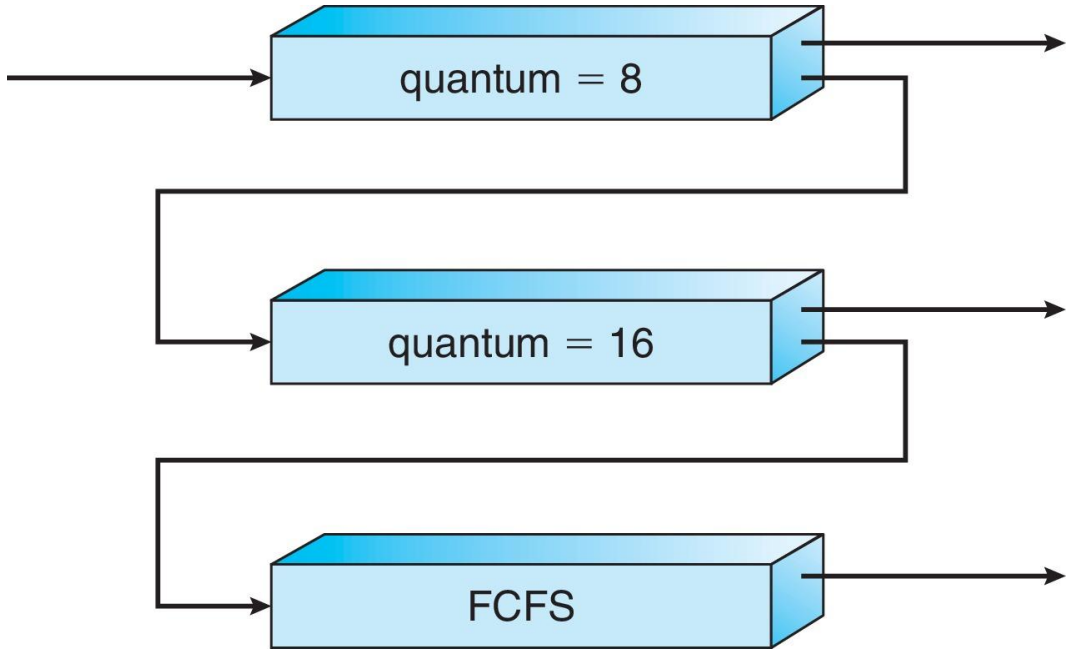
Multilevel Feedback Queue

- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS



Scheduling

- A new process enters queue Q_0 which is served in RR
  - When it gains CPU, the process receives 8 milliseconds
  - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
- At Q_1 job is again served in RR and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads are supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Multiple-Processor Scheduling

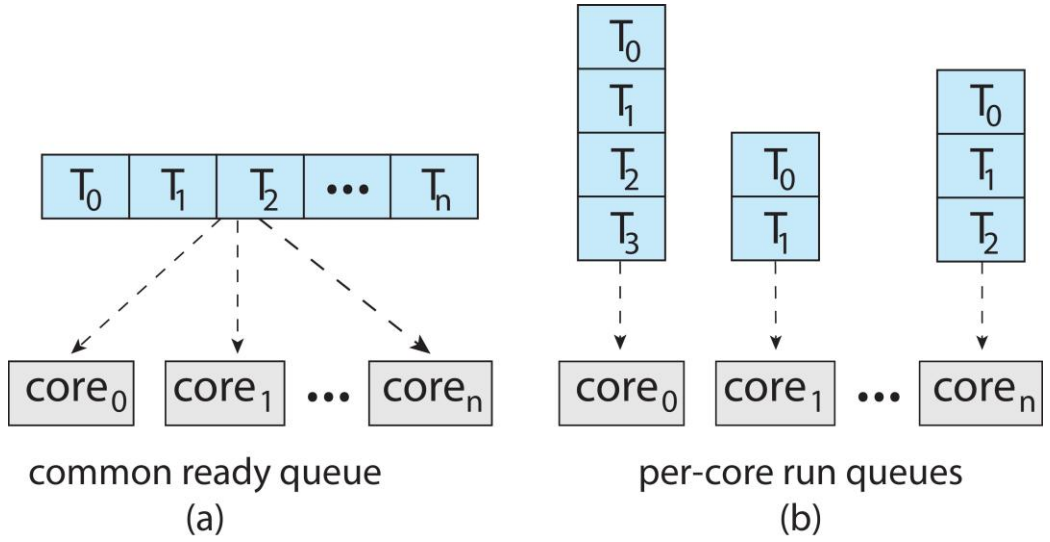
- CPU scheduling is more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems

Multiple-Processor Scheduling

Symmetric multiprocessing (SMP) is where each processor is self scheduling.

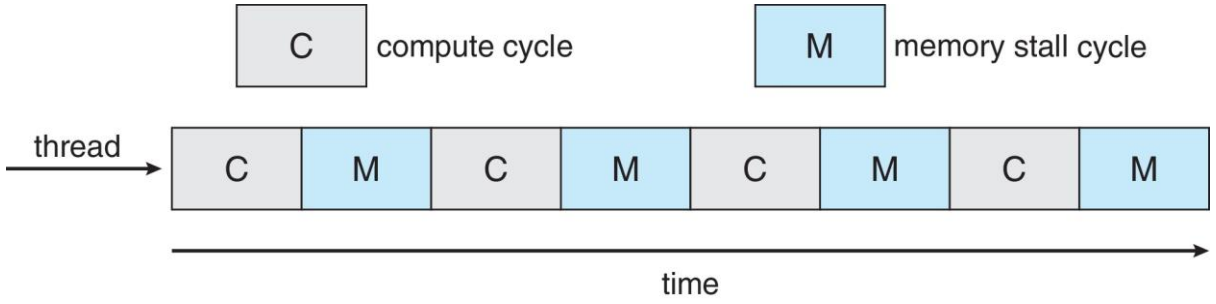
All threads may be in a common ready queue (a)

Each processor may have its own private queue of threads (b)



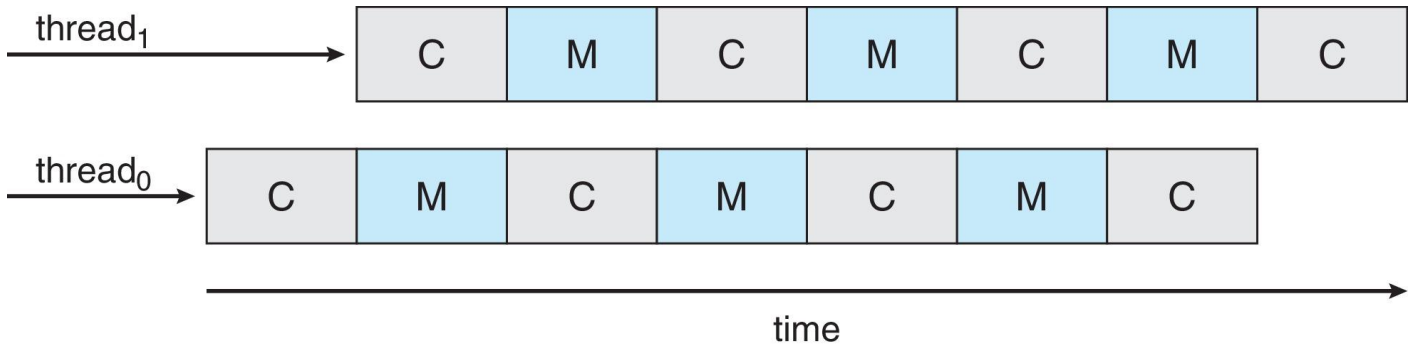
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieval happens



Multithreaded Multicore System

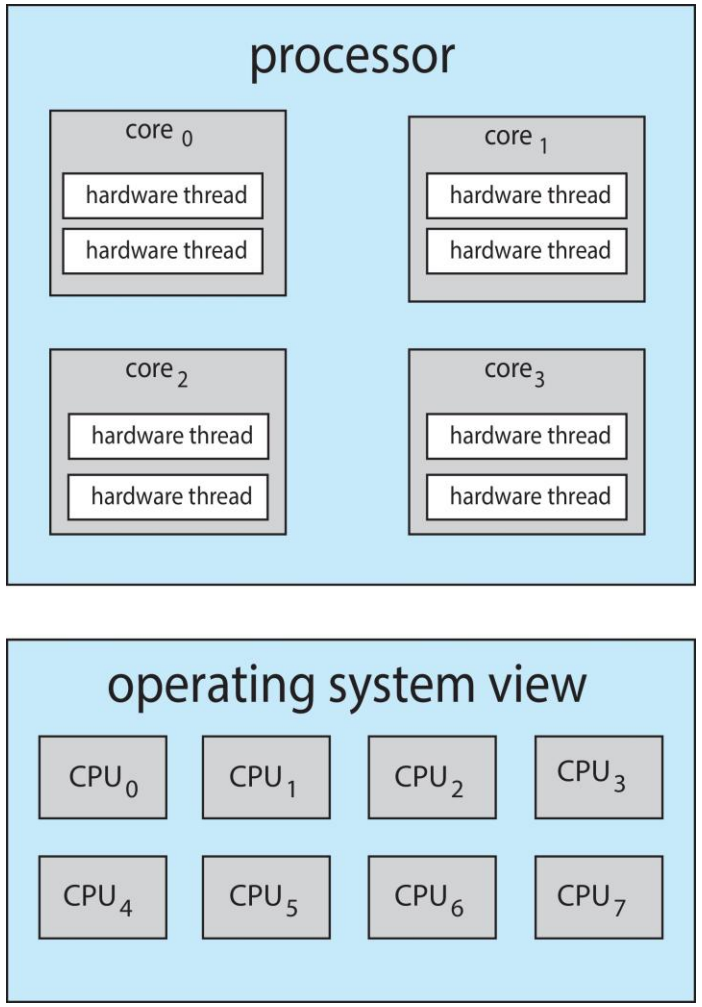
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns multiple hardware threads to each core. (Intel refers to this as **hyperthreading**.)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

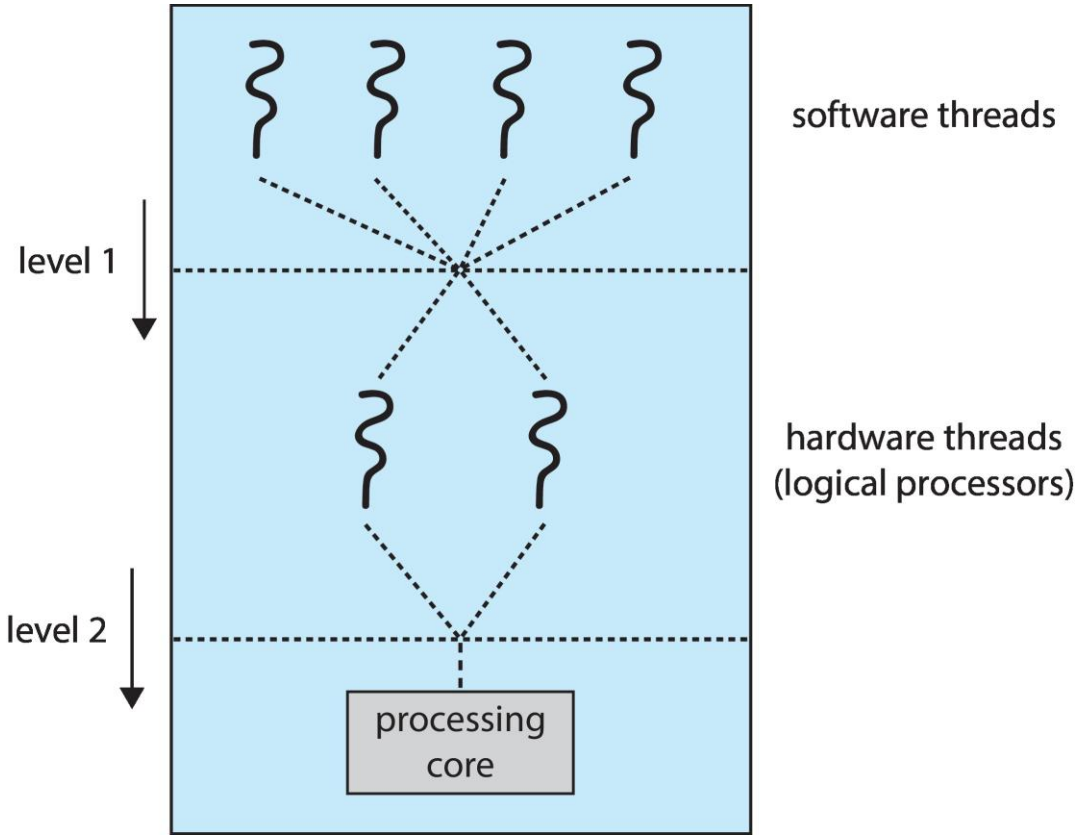


Multithreaded Multicore System

Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.



Multiple-Processor Scheduling – Load Balancing

- If SMP, needs to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep the workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pull waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

When a thread has been running on one processor, the cache contents of that processor store the memory accessed by that thread.

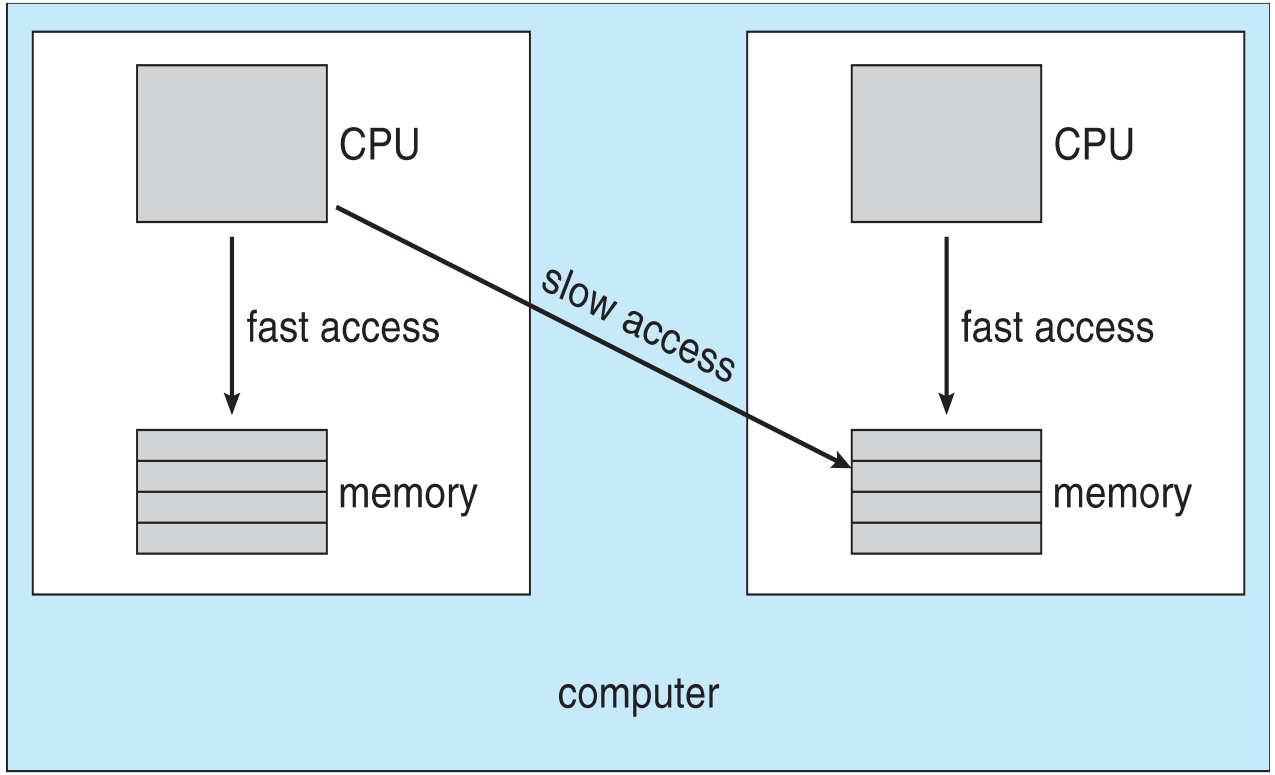
We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)

Load balancing may affect processor affinity. A thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.

Hard affinity – allows a process to specify a set of processors it may run on. (e.g. Linux)

If the operating system is **NUMA-aware**, it will assign memory close to the CPU the thread is running on.



NUMA and CPU Scheduling

Real-Time CPU Scheduling

Can present obvious challenges

- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**

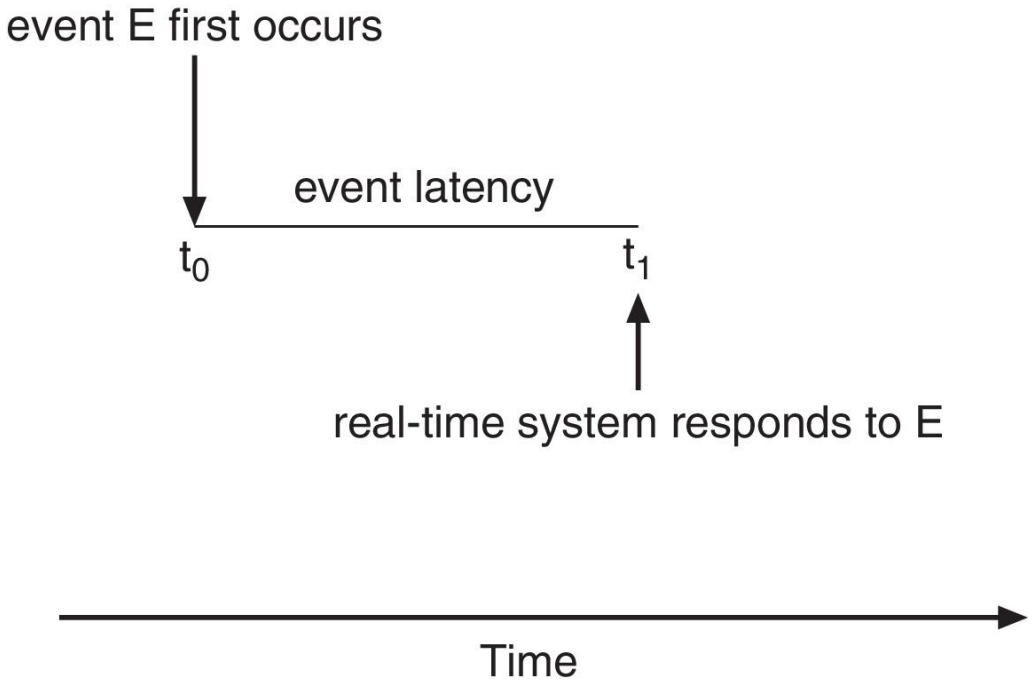
Real-Time CPU Scheduling

Event latency – the amount of time that elapses from when an event occurs to when it is serviced.

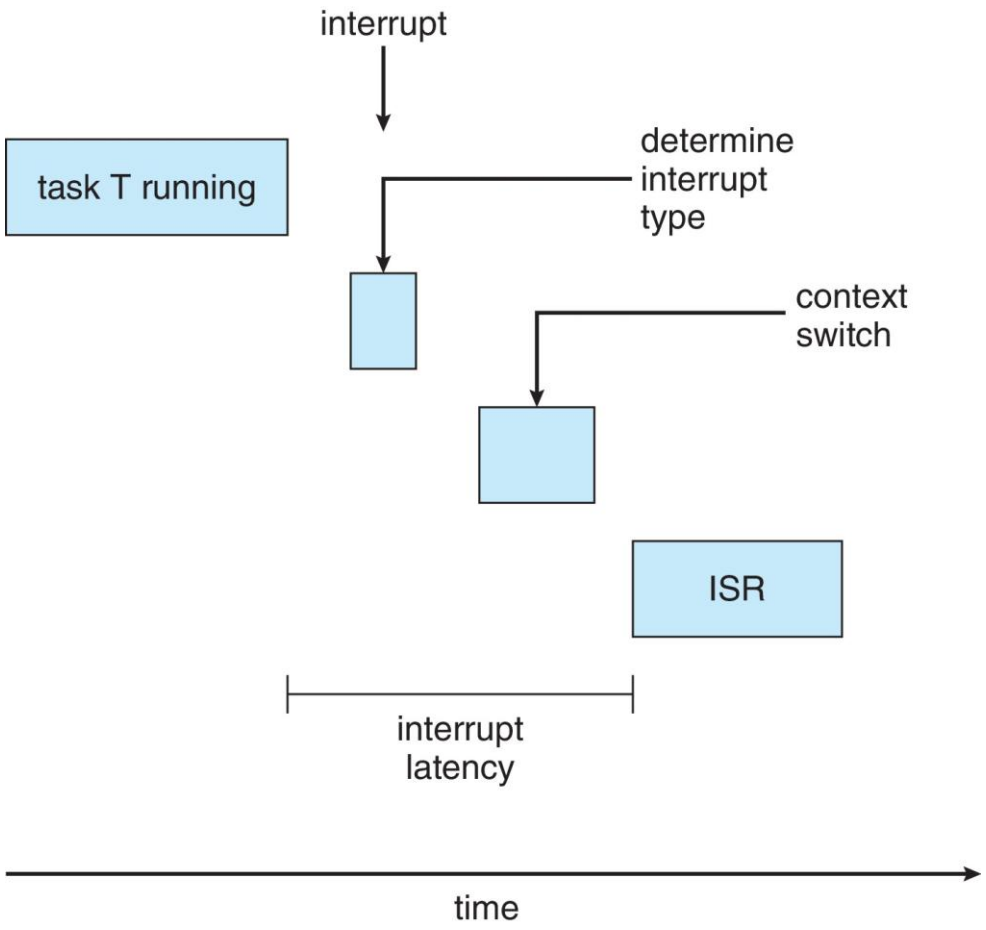
Two types of latencies affect performance

1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt

2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another



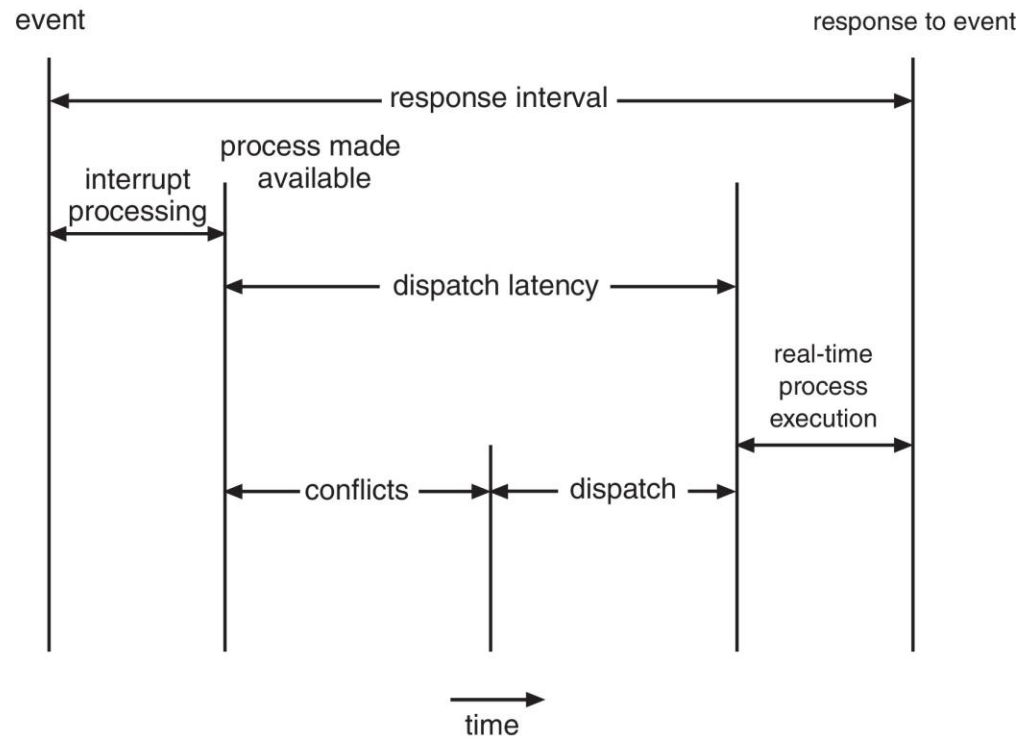
Interrupt Latency



Dispatch Latency

Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

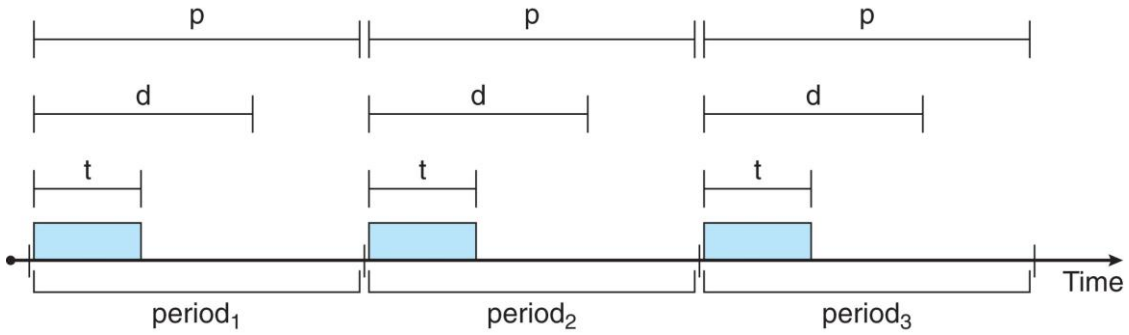
For real-time scheduling, scheduler must support preemptive, priority-based scheduling

- But only guarantees soft real-time

For hard real-time must also provide ability to meet deadlines

Processes have new characteristics: **periodic** ones require CPU at constant intervals

- Has processing time *t*, deadline *d*, period *p*
- $0 \leq t \leq d \leq p$
- **Rate** of periodic task is $1/p$



POSIX Real-Time Scheduling

The POSIX.1b standard

API provides functions for managing real-time threads

Defines two scheduling classes for real-time threads:

1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority

2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority

Defines two functions for getting and setting scheduling policy:

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`

2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

Linux Scheduling Through Version 2.5

- Before kernel version 2.5, ran a variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
- Each has specific priority
- Scheduler picks highest priority task in highest scheduling class
- Rather than quantum based on fixed time allotments, based on proportion of CPU time
- Two scheduling classes included, others can be added
    - Default (CFS)
    - real-time

Linux Scheduling in Version 2.6.23 + (Cont.)

Quantum calculated based on **nice value** from -20 to +19

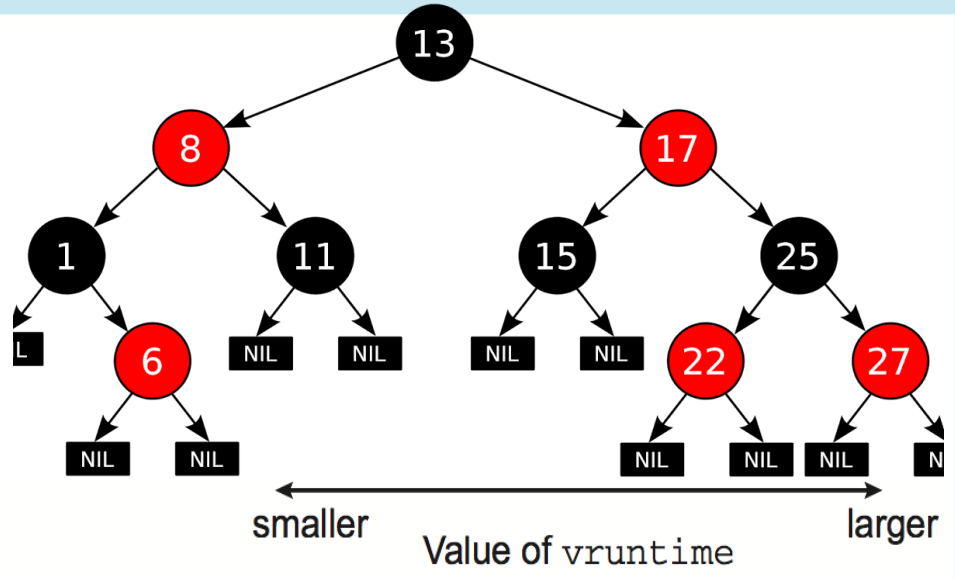
- Lower value is higher priority
- Calculates **target latency** – interval of time during which task should run at least once
- Target latency can increase if say number of active tasks increases

CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**

- Associated with decay factor based on priority of task – lower priority is higher decay rate
- Normal default priority yields virtual run time = actual run time

To decide next task to run, scheduler picks task with lowest virtual run time

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

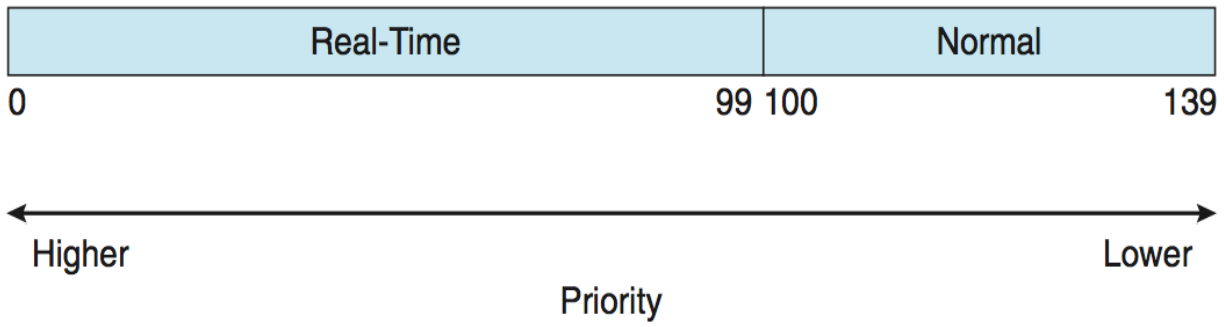


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg\,N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

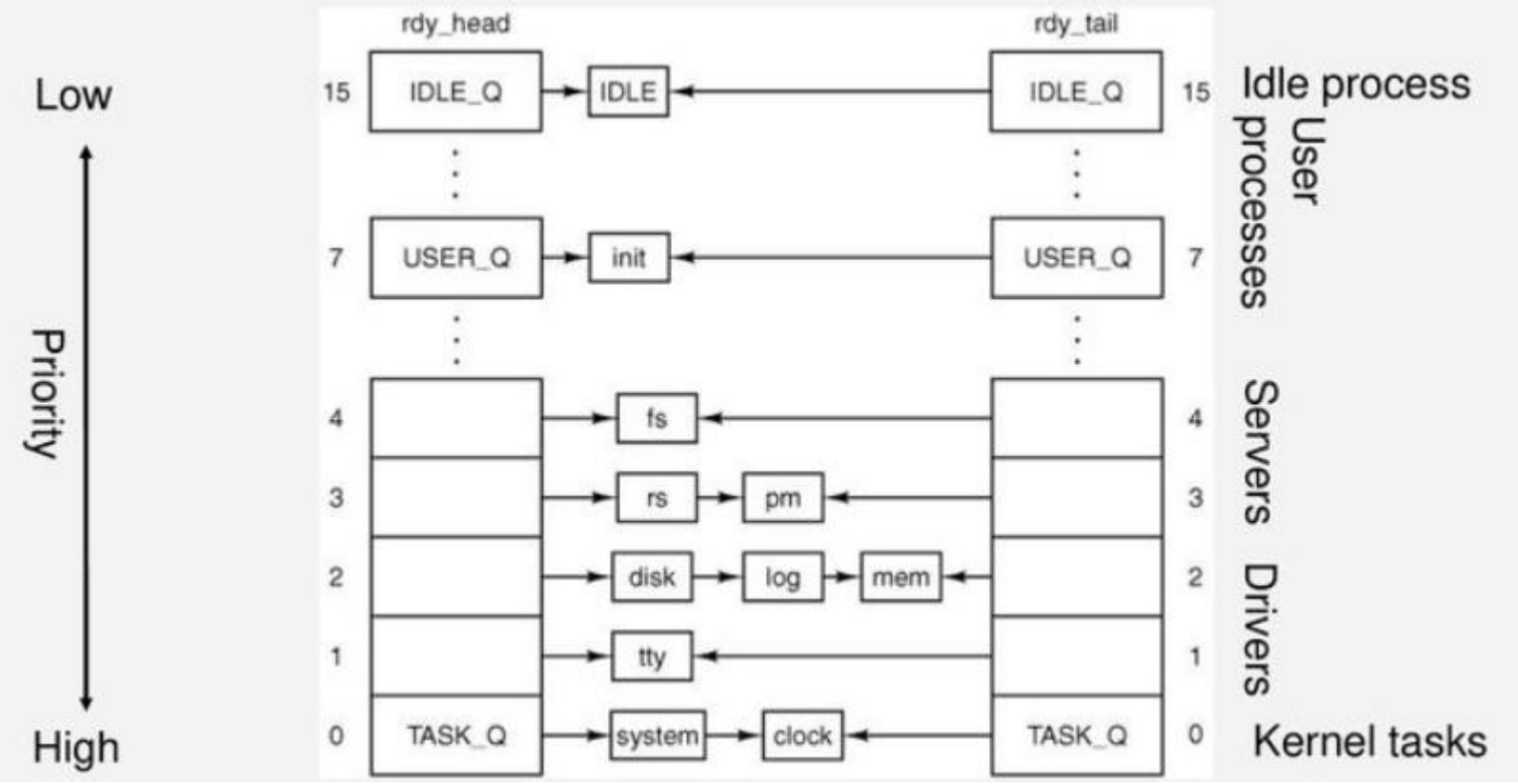
CFS Performance

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Scheduling in Minix



Thank you

Juan F. Medina

Juan.medina26@upr.edu

Memory management

Juan Felipe Medina Lee, Ph.D.

Computer Science and Engineering
University of Puerto Rico
Mayagüez Campus

Outline

- Background
- Contiguous Memory Allocation
- Paging
- Swapping

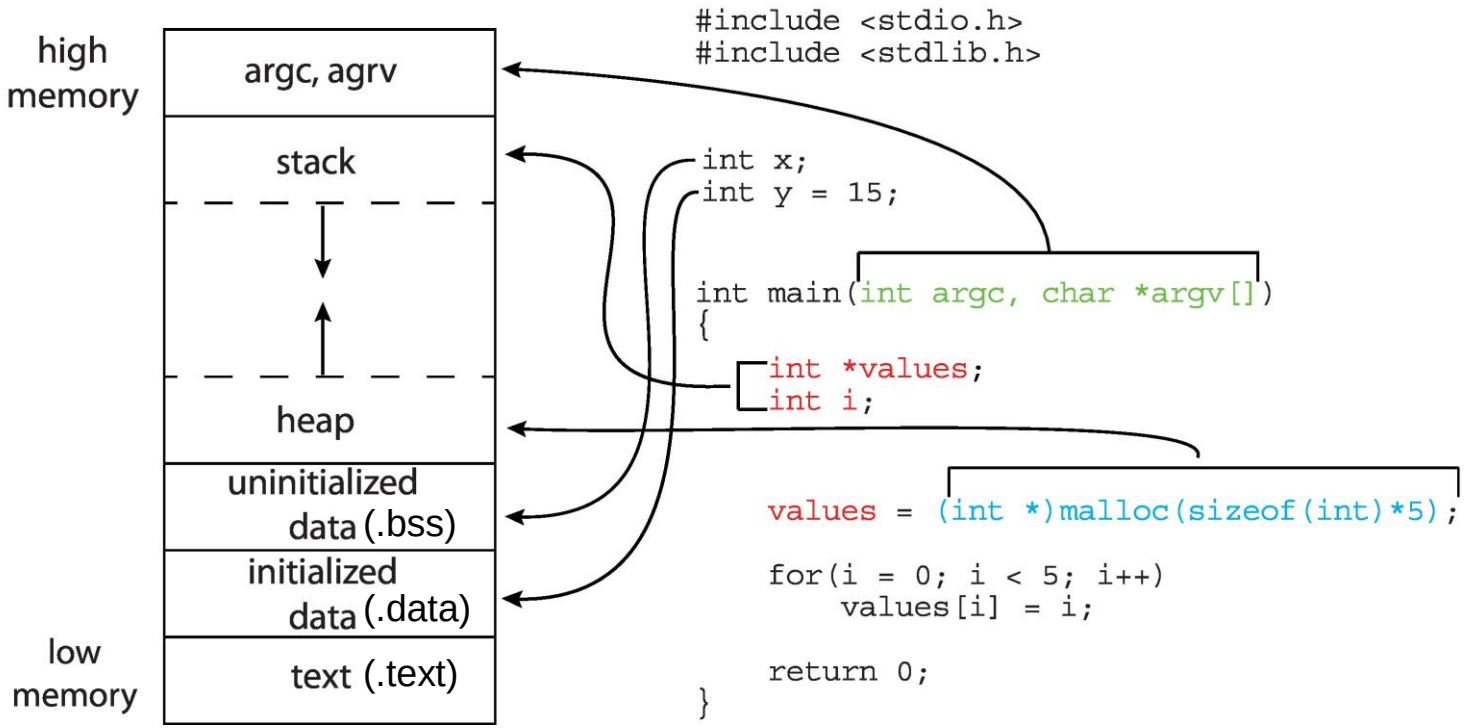
Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques.
- To discuss a practical example in Linux of memory management.

Background

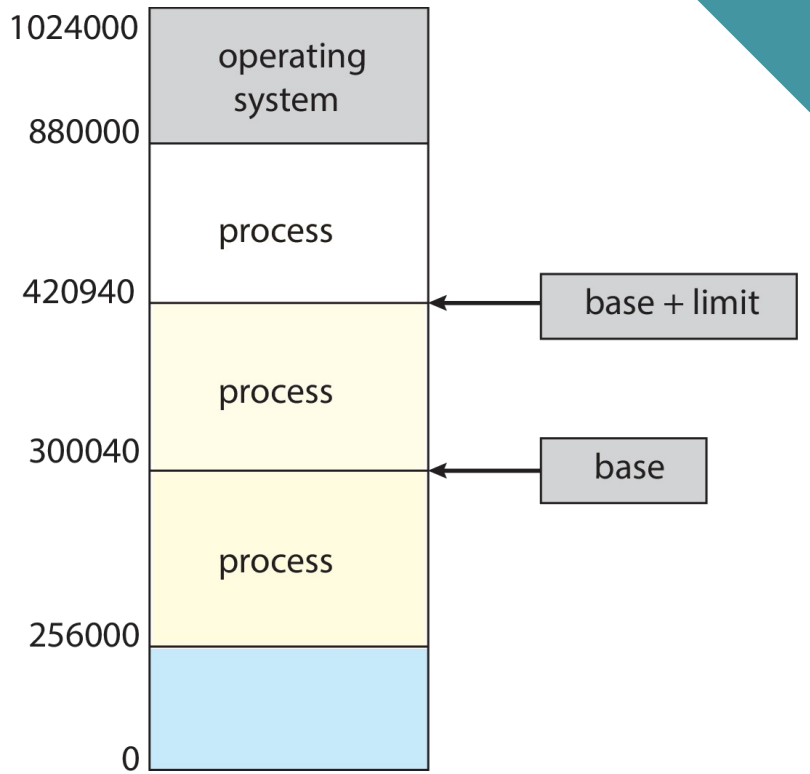
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Memory Layout of a C Program



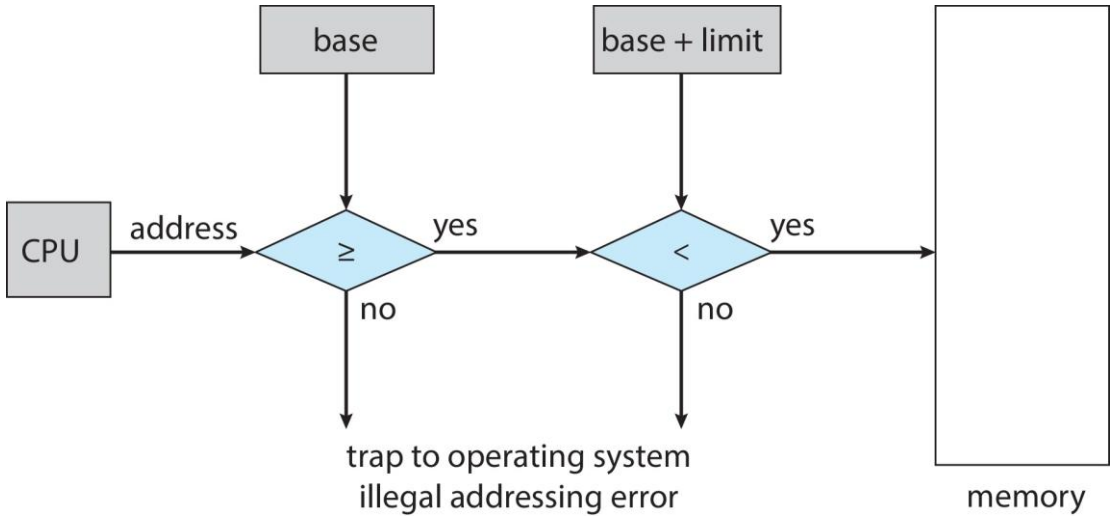
Protection

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The instructions to loading the base and limit registers are privileged

Address Binding

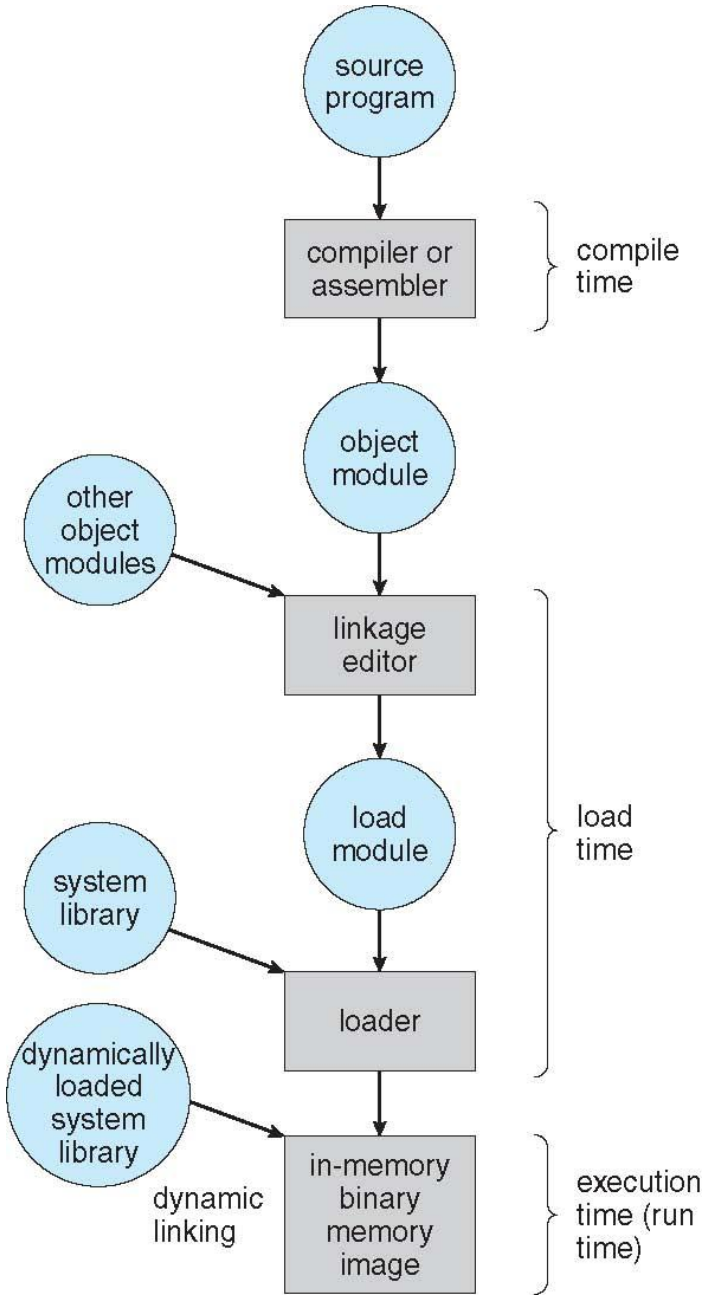
- Programs on disk, ready to be brought into memory to execute from an **input queue**
- As the process executes, it accesses instructions and data from memory.

- Addresses are represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from the beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - **Each binding maps one address space to another**

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location is known a priori, **absolute code** can be generated; must recompile code if the starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
  - Need hardware support for address maps (e.g., base and limit registers)



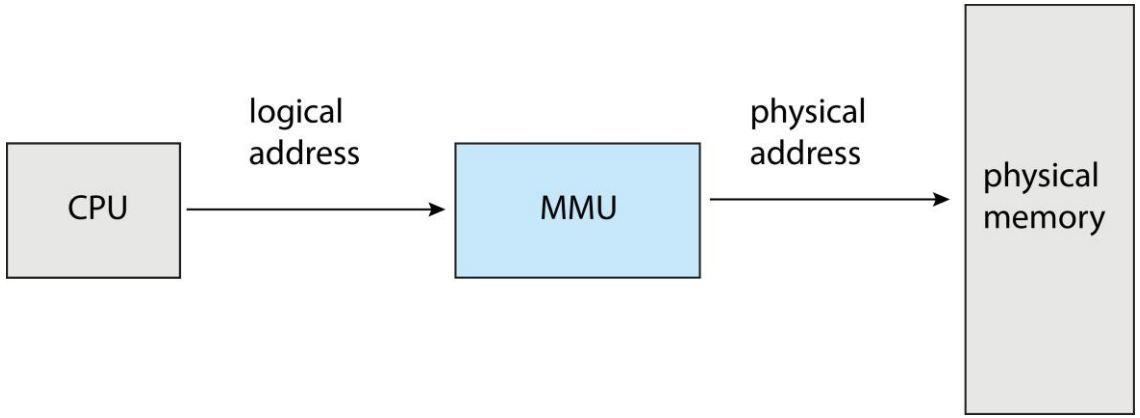
Multistep Processing of a User Program

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme
  - **Logical address space** is the set of all logical addresses generated by a program
  - **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

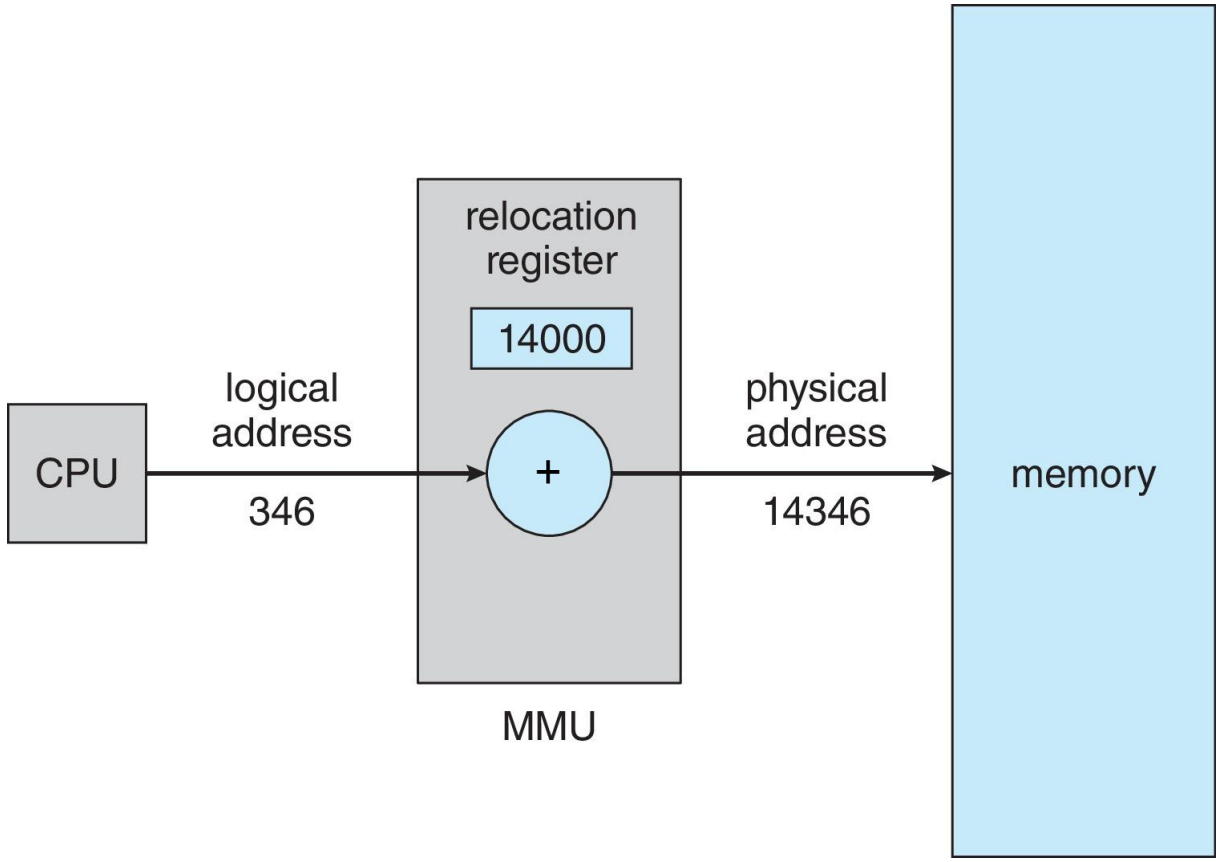
The hardware device that at run time **maps virtual to a physical address**



Memory-Management Unit (Cont.)

- Consider a simple scheme. A generalization of the base-register scheme.
- The base register now called the **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Memory-Management Unit (Cont.)



Dynamic Loading

- The entire program does not need to be in memory to execute
- **A routine is not loaded until it is called**
- Better memory-space utilization; unused routine is never loaded
- All routines are kept on disk in a relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases

Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image (**very inefficient**)
- **Dynamic linking** –linking postponed until execution time

- The operating system checks if the routine is in the processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- Also known as **shared libraries**

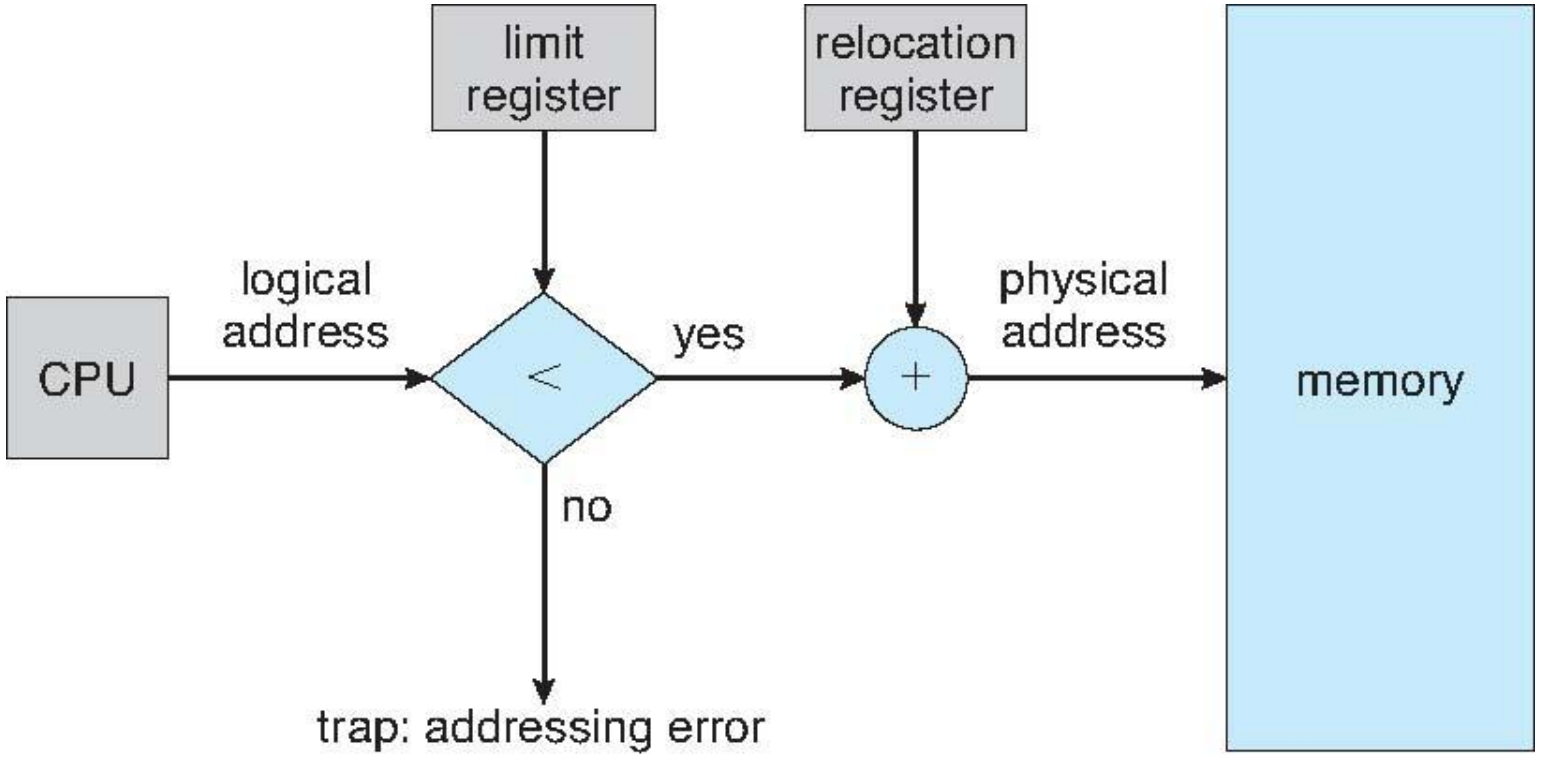
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resources and must be allocated efficiently
- Contiguous allocation is one early method
- Main memory is usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in a single contiguous section of memory

Contiguous Allocation (Cont.)

Relocation registers are used to protect user processes from each other and from changing operating-system code and data

- **Base register** contains the value of the smallest physical address
- **Limit register** contains a range of logical addresses – each logical address must be less than the limit register
- **MMU** maps logical addresses *dynamically*
- Can then allow actions such as kernel code being **transient** and kernel changing size

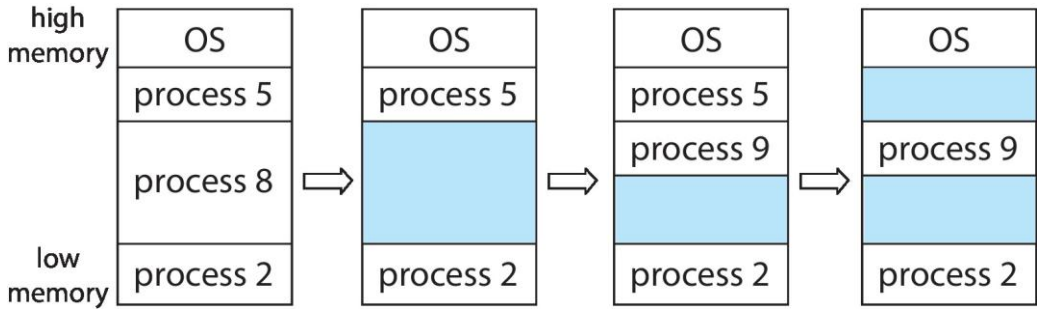


Hardware support for relocation and limit registers

Variable Partition

Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable partition** sizes for efficiency (sized to a given process's needs)
- **Hole** – block of available memory; holes of various sizes are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- The operating system maintains information about:
  - allocated partitions
  - free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

First-fit: Allocate the ***first*** hole that is big enough

Best-fit: Allocate the ***smallest*** hole that is big enough; must search the entire list, unless ordered by size

- Produces the smallest leftover hole

Worst-fit: Allocate the ***largest*** hole; must also search the entire list

- Produces the largest leftover hole

- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization, respectively.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition but not being used

- First fit analysis reveals that given *N* blocks allocated, **0.5 *N* blocks lost to fragmentation**
    - **50-percent rule**

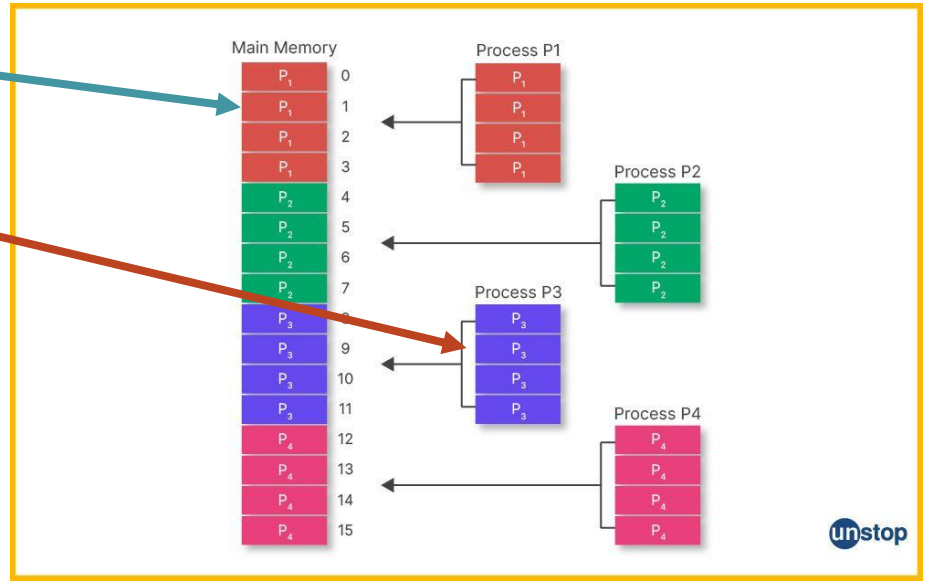
Fragmentation (Cont.)

You can reduce external fragmentation by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is **possible only if relocation is dynamic** and occurs at execution time.
- Relocation requires only moving the program and data, and then changing the base register to reflect the new base address.

Paging

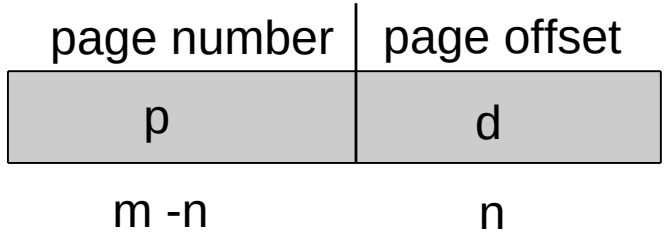
- **The physical address space of a process can be noncontiguous; the process is allocated physical memory whenever the latter is available**
  - Avoids external fragmentation
  - Avoids problem of varying-sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 MB
- Divide logical memory into blocks of the same size called **pages**
- Keep track of all free frames
- To run a program of size *N* pages, you need to find *N* free frames and load the program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation



Address Translation Scheme

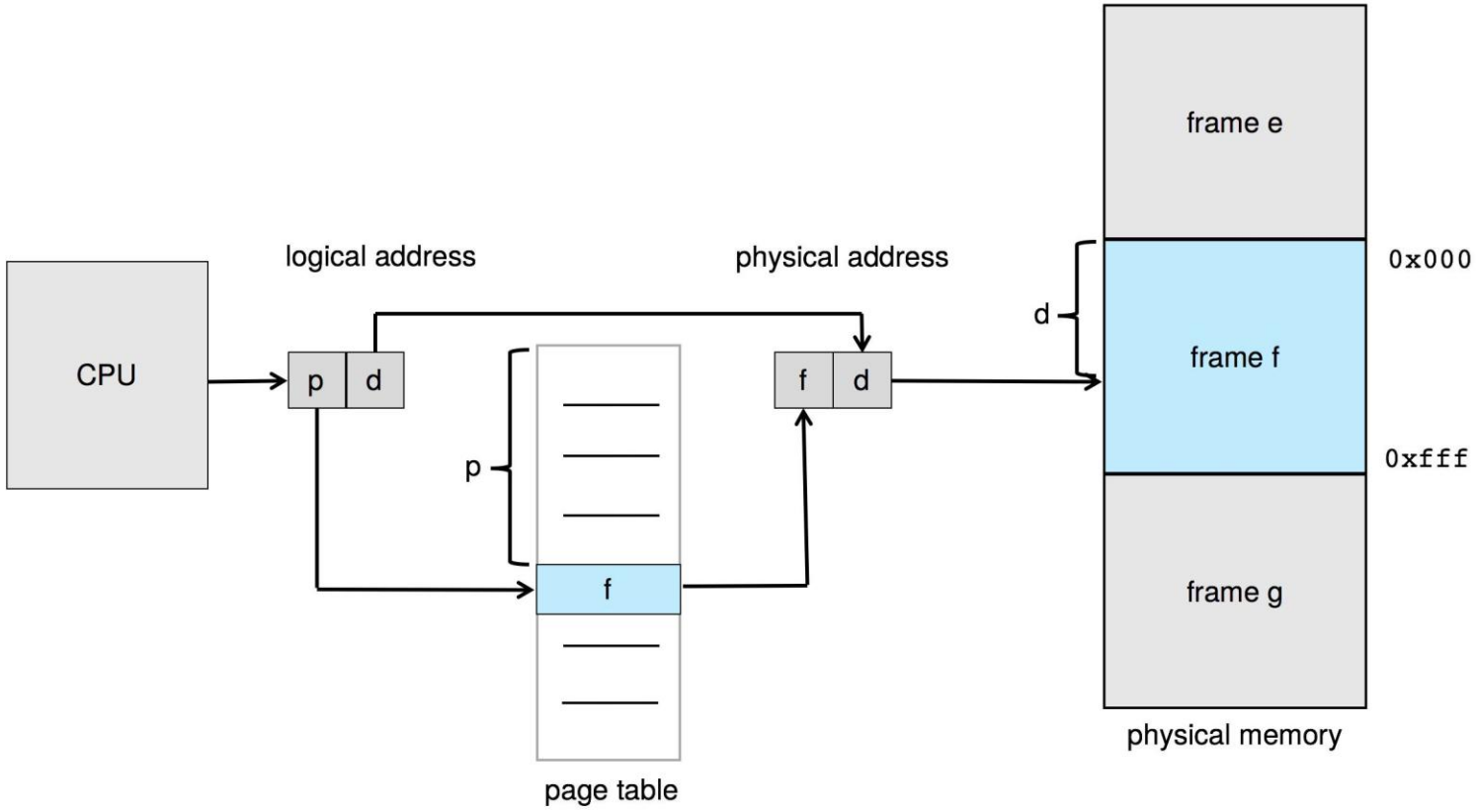
Address generated by CPU is divided into:

- **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
- **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

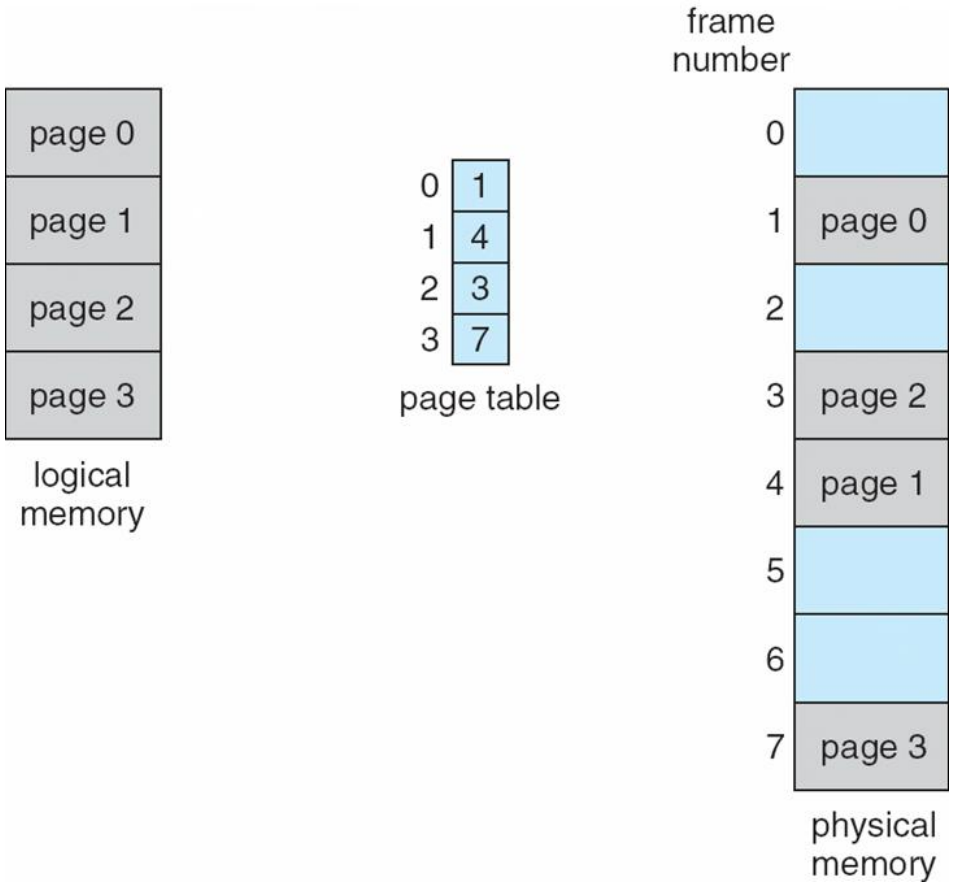


- For given logical address space 2^m and page size 2^n

Paging Hardware



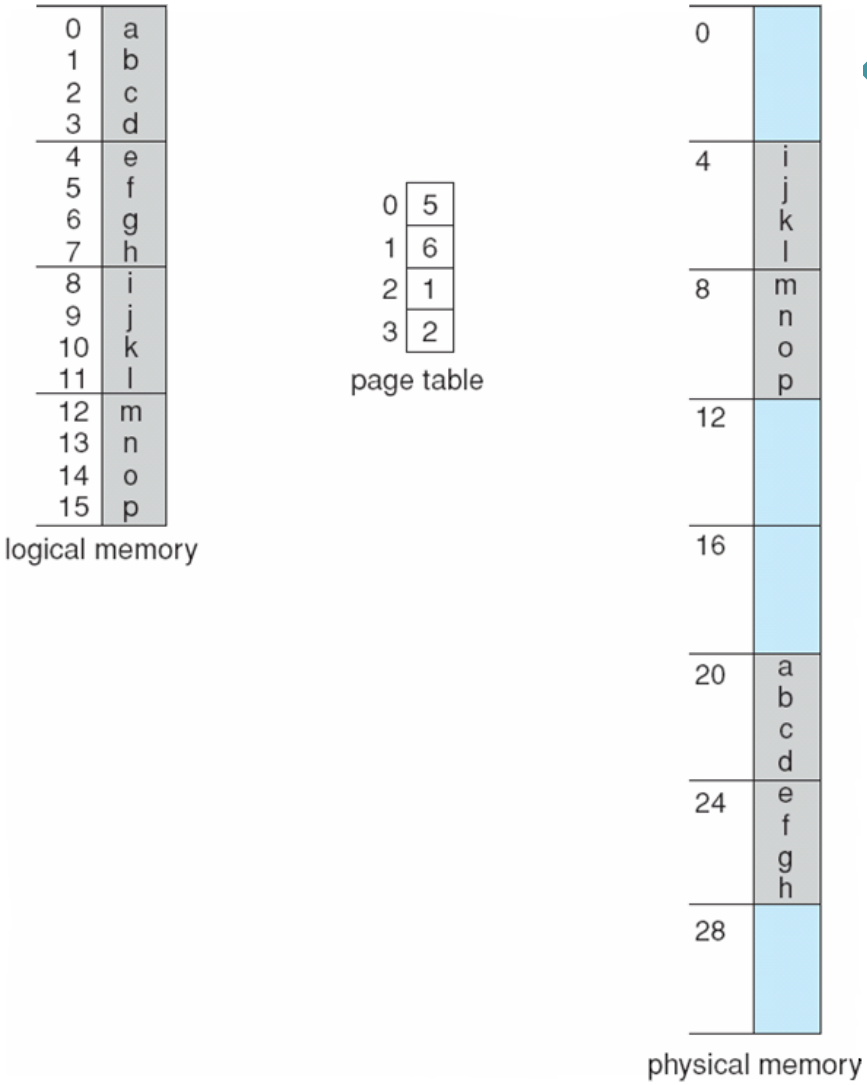
Paging Model of Logical and Physical Memory



Paging Example

- What is the page size?
- What is the logical address space size?
- What are the values of n and m?
  - m: bits to address the logical address space
  - n: bits to access bytes within a page

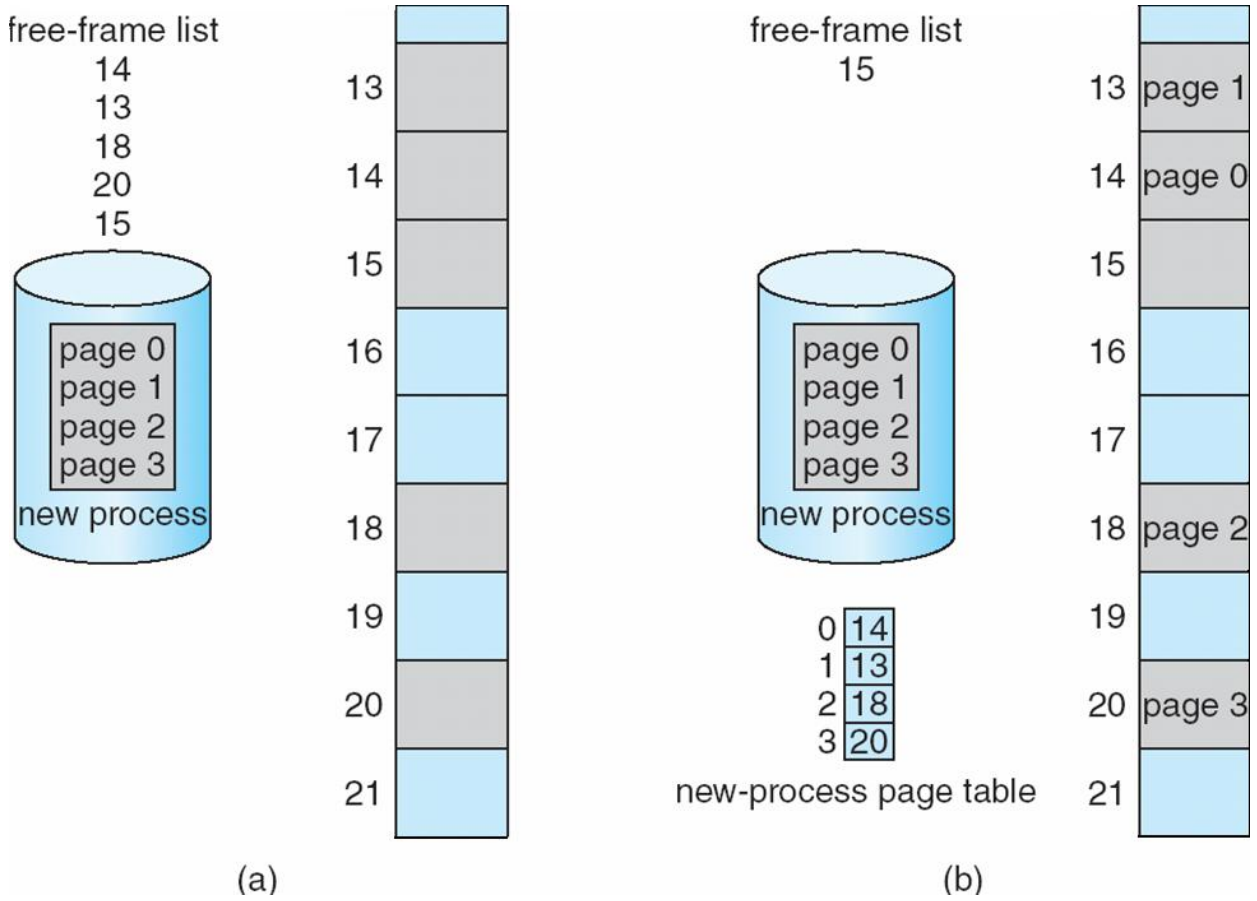
- Logical address: n = 2 and m = 4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes, Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- Are small frame sizes desirable?
  - Each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes: 8 kB and 4 MB
  - Ubuntu has default page size of 4kB

Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Implementation of Page Table

Page table is kept in main memory

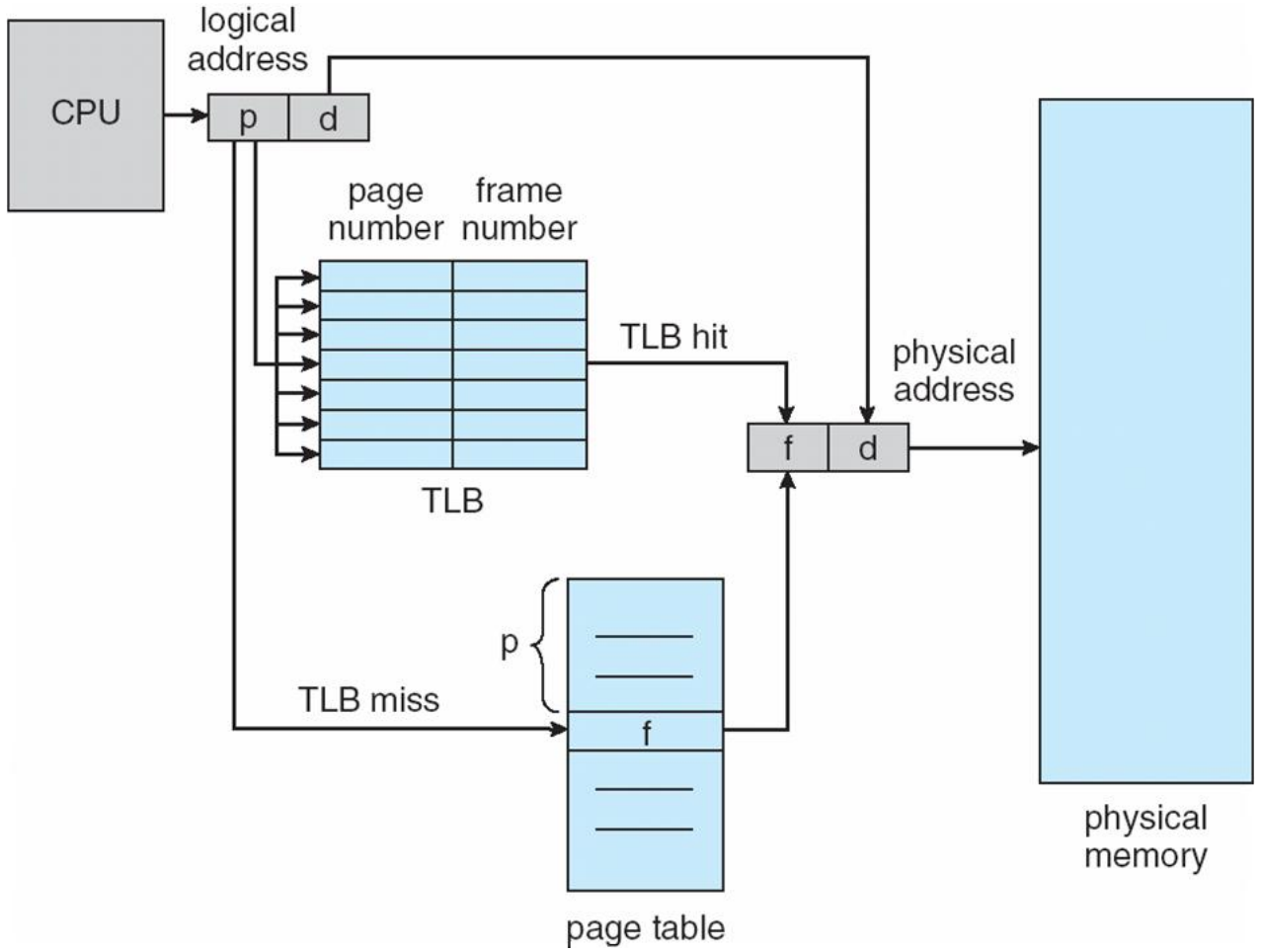
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table

In this scheme, every data/instruction access requires two memory accesses

- One for the page table and one for the data/instruction

The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

Paging Hardware With TLB



Memory Protection

Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed

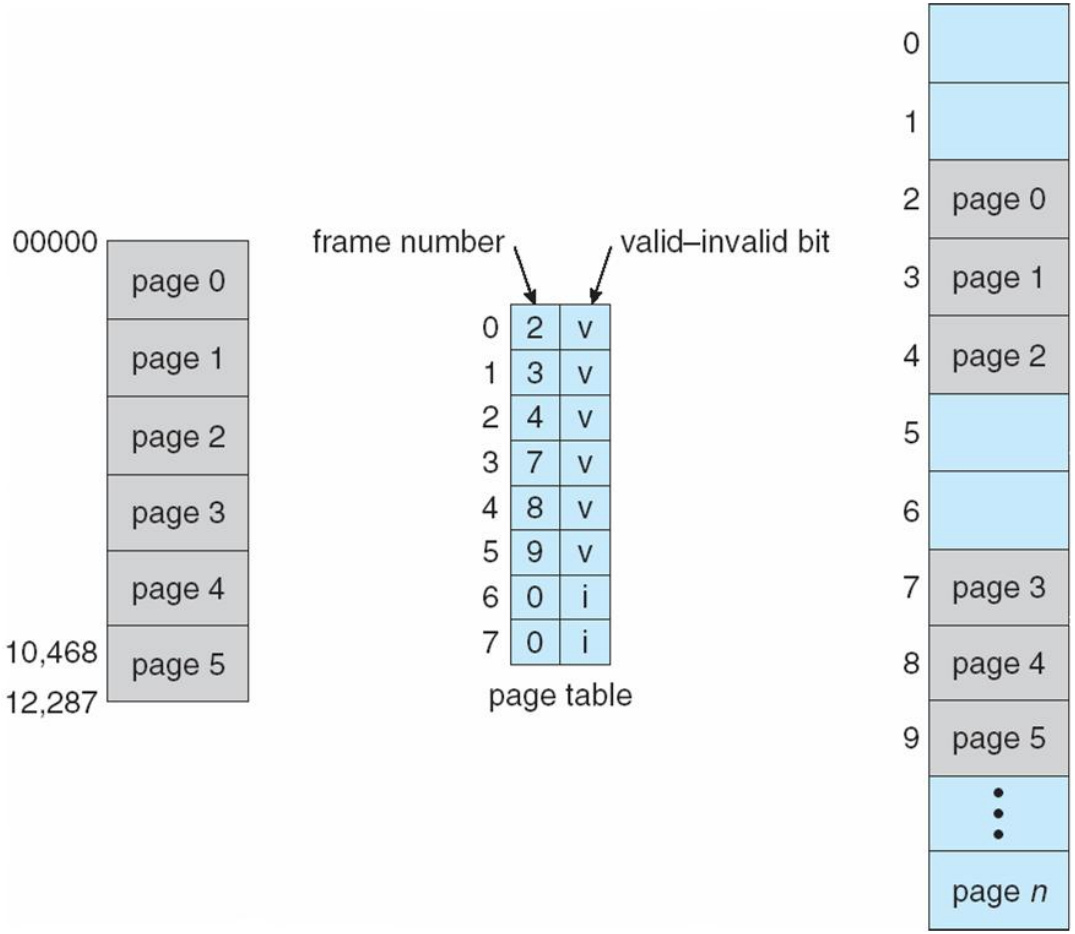
- Can also add more bits to indicate page execute-only, and so on

Valid-invalid bit attached to each entry in the page table:

- “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
- “invalid” indicates that the page is not in the process’ logical address space
- Or use **page-table length register** (**PTLR**)

Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table



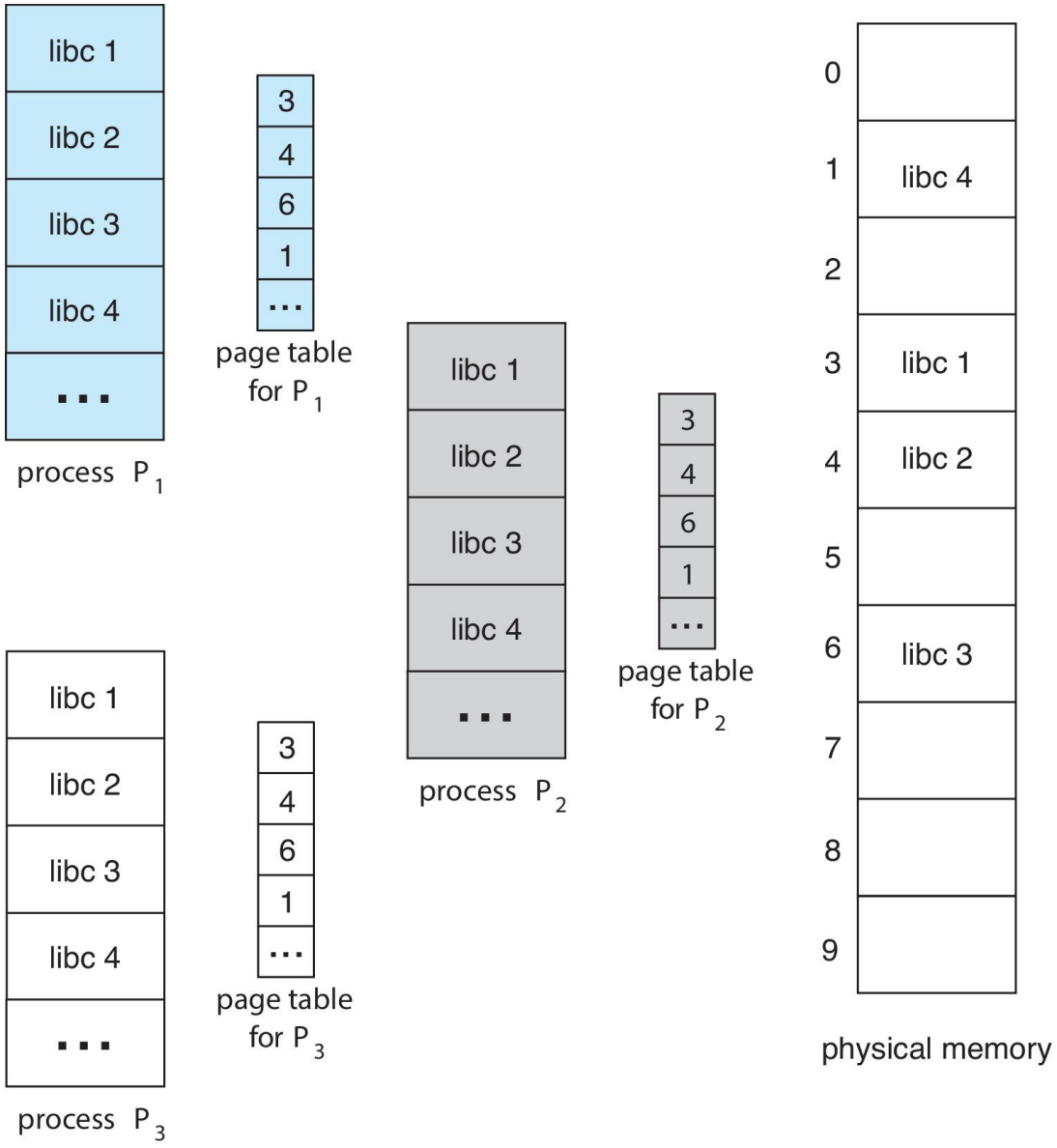
Shared Pages

Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

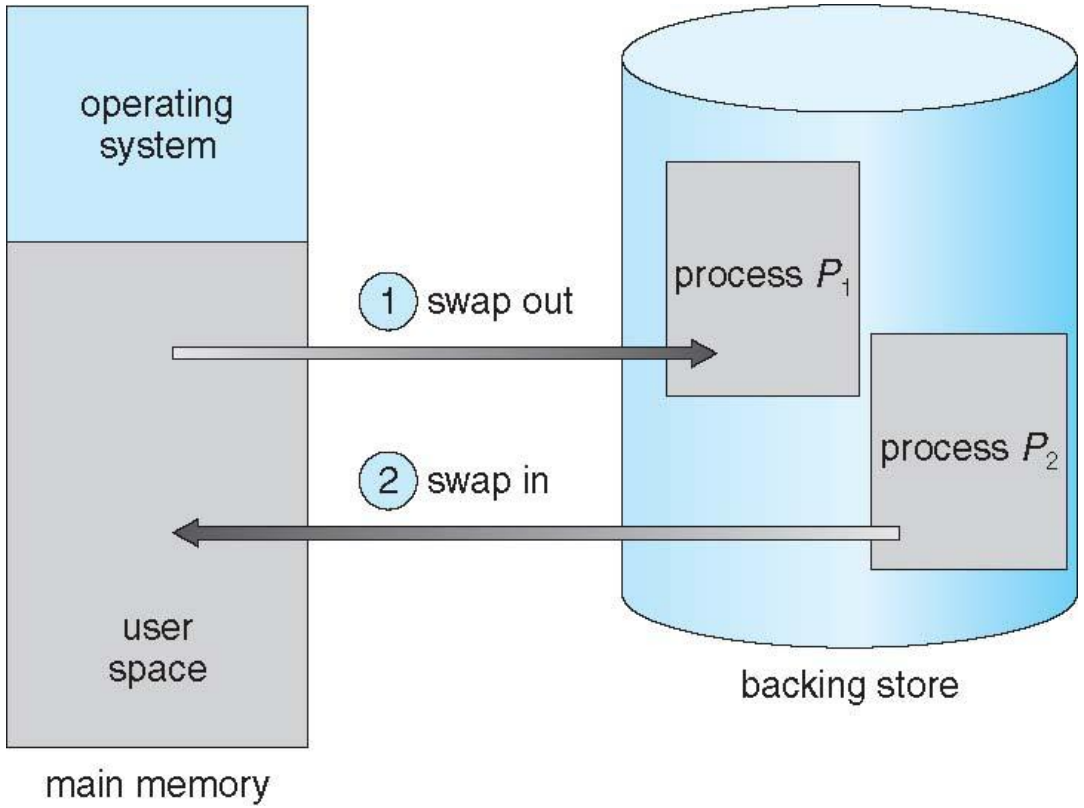


Shared Pages Example

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
- The total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- A major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes that have memory images on disk

Schematic View of Swapping



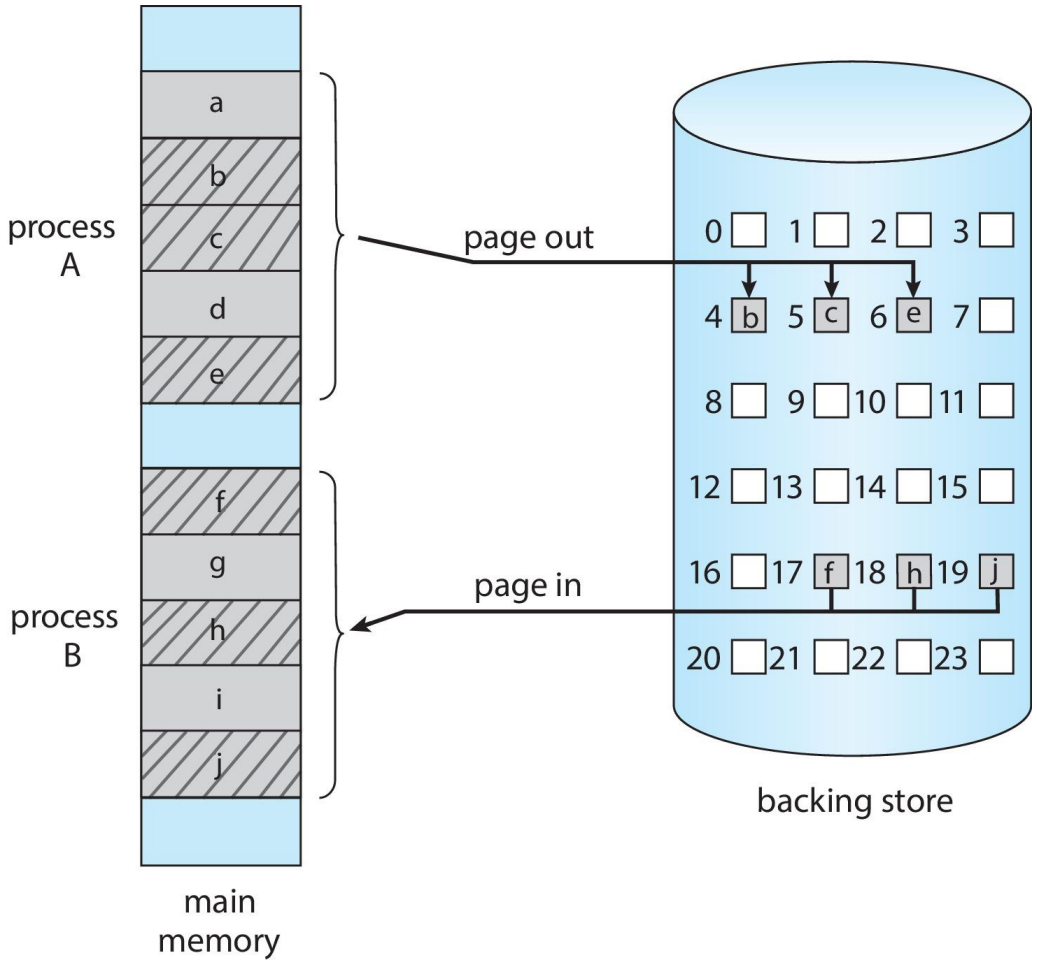
Swapping (Cont.)

- Does the swapped-out process need to swap back into the same physical addresses?
- Depends on address binding method
  - Plus, consider pending I/O to/from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping is normally disabled
  - **Started if more than the threshold amount of memory is allocated**
  - **Disabled again once memory demand reduced below the threshold**

Context Switch Time including Swapping

- If next process to be put on the CPU is not in memory, need to swap out a process and swap in the target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - Swap out time of 2000 ms
    - Plus swap in of same-sized process
    - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
    - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Swapping with Paging



Thank you

Juan F. Medina

Juan.medina26@upr.edu