Operating System Principles - 2

Shyan-Ming Yuan
CS Department, NYCU
smyuan@gmail.com

Life Cycle of a Thread/Process

- Thread (Process) execution consists of a cycle of
  - CPU execution and I/O wait.
  - They are called CPU burst and I/O burst, respectively.
- Overall system throughput can be increased if some threads are in their CPU burst while others are waiting for I/O.
- A scheduler is responsible for deciding what thread should run next.
  - If the current thread did not finish its CPU burst but it has run too long, the scheduler may preempt the current thread.
- If a scheduler can preempt a thread and context switch to another thread then it is a preemptive scheduler.
  - Otherwise it is a non-preemptive.

CPU Scheduler & Dispatcher

- The CPU scheduler selects a thread from threads in memory that are ready to execute, and allocates the CPU to it.
  - CPU scheduling decisions may take place when a thread:
    1. Switches from running to waiting state (non-preemptive/system call)
    2. Switches from running to ready state (preempted by time expiration)
    3. Switches from waiting to ready (the running thread may then be preempted by a higher-priority thread)
    4. Terminates (non-preemptive)
- The dispatcher gives control of the CPU to the thread selected by the CPU scheduler. It includes the following actions:
  - switching context, switching to user mode and jumping to the proper location in the user program to restart that thread.
- Dispatch latency is the time it takes for the dispatcher to stop one thread (process) and start running another.

Scheduling Criteria

- CPU utilization is the percentage of CPU being busy
- Throughput is the number of threads (processes) that complete their execution per time unit
- Turnaround time is the total latency for executing a particular thread (process)
- Waiting time is the total amount of time that a thread (process) has been waiting in the ready queue
- Response time is the latency that a thread (process) takes from accepting a request until produces the first response (for time-sharing environment)
- Time slice (or time quantum) is the maximum time a thread (process) can run before being preempted.

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

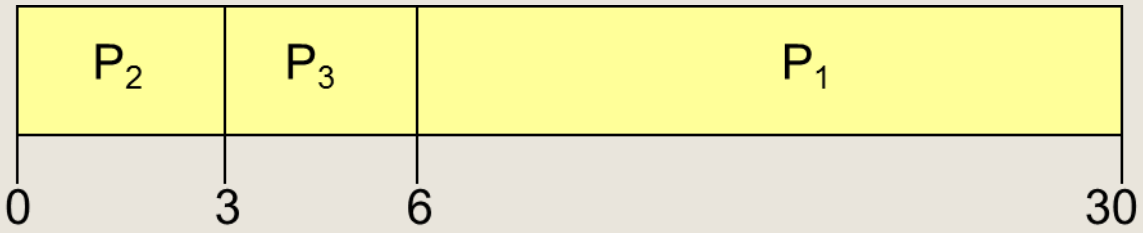
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont)

- When 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: $(6 + 0 + 3)/3 = 3$
  - It is much better than the previous case
- Convoy effect :
  - A short process may be slow down by a long process

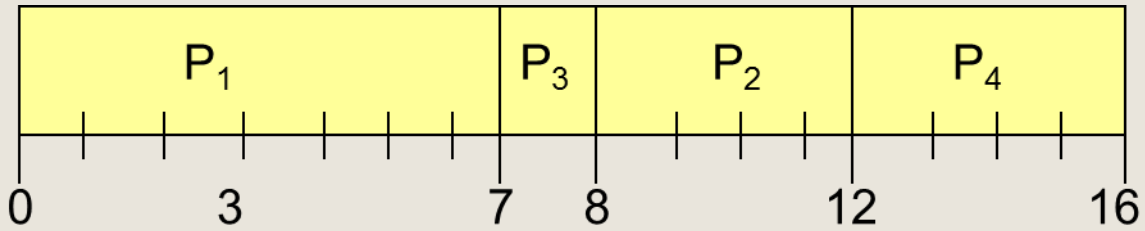
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
  - Use these lengths to schedule the shortest process first
- SJF is optimal with respective to average waiting time for a given set of processes.
  - The difficulty is knowing the length of the next CPU request

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

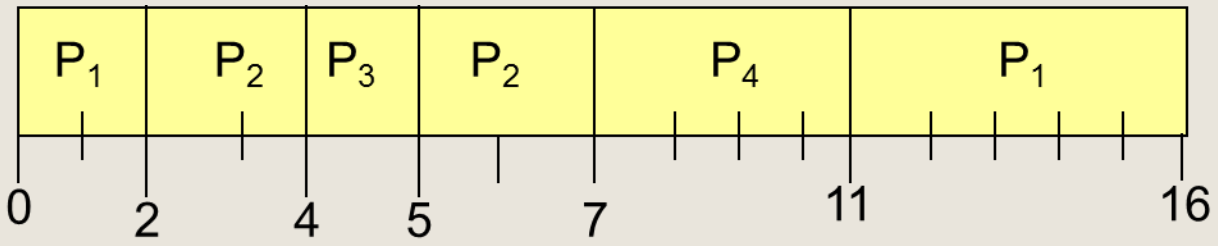


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Shortest Remaining Time First (SRTF) (Preemptive SJF)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SRTF (preemptive SJF)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

The Next CPU Burst

- A typical mechanism to estimate the length of next CPU burst is:
  - A weighted exponential average of previous CPU bursts can be used.
  - The weight may be adjusted to weigh recent CPU bursts more or less than historical averages.

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

$$\tau_{n+1} = \alpha t_n + (1-\alpha)(\alpha t_{n-1} + (1-\alpha)\tau_{n-1}) = \alpha t_n + \alpha(1-\alpha)t_{n-1} + (1-\alpha)^2\tau_{n-1}$$

$$\tau_{n+1} = \alpha\left(\sum_{i=0}^{n}(1-\alpha)^i t_{n-i}\right), where\ \tau_0 = 0$$

Round Robin (RR)

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

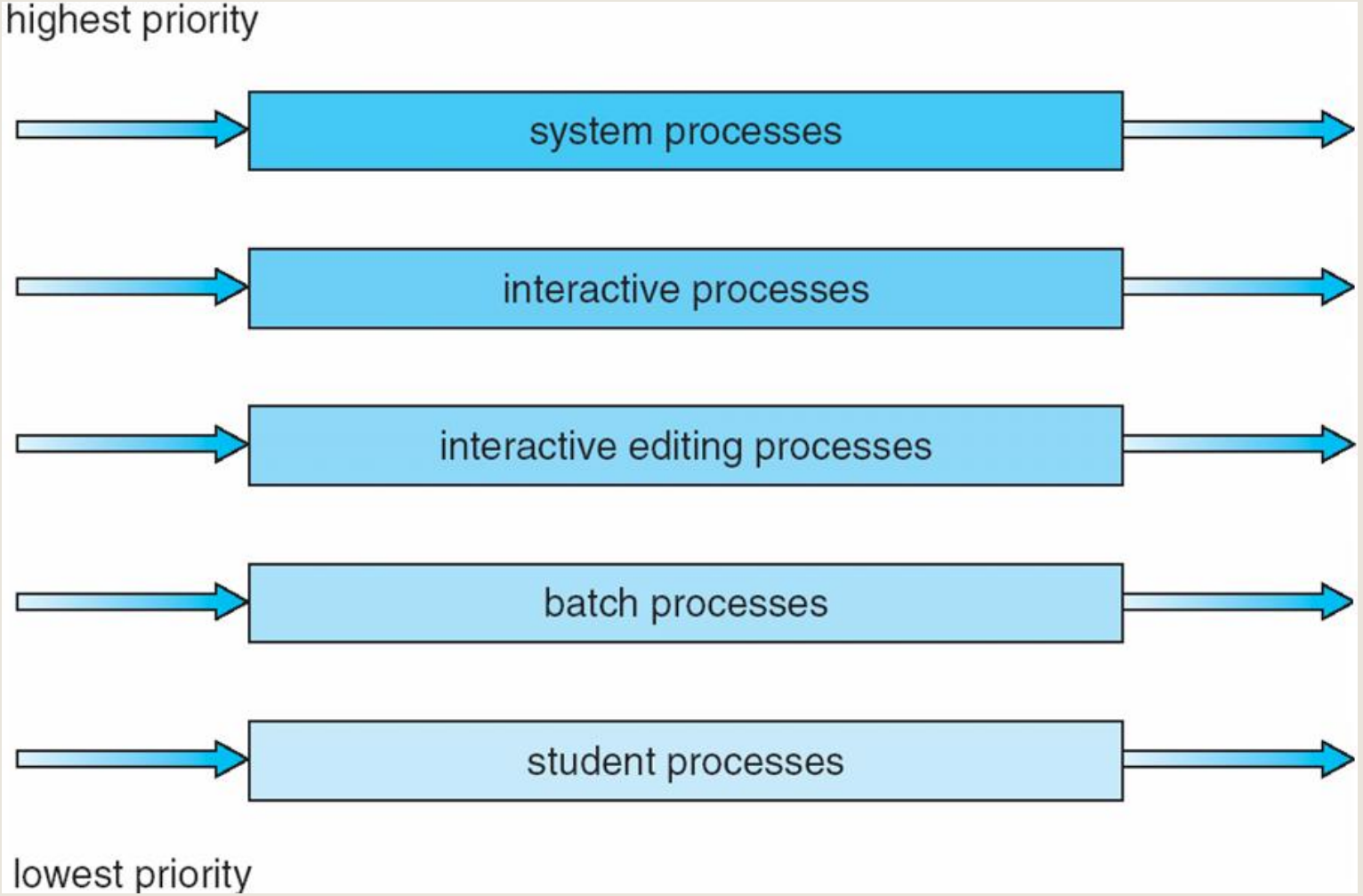
Priority Scheduling

- A priority number (integer) is associated with each thread
- The CPU is allocated to the thread with the highest priority
  - It can be either preemptive or nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority threads may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of each thread

Multilevel Queues

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling between the queues
  - Fixed priority scheduling – serve all processes from foreground queue then background queue (May lead to starvation)
  - Time slice – each queue gets a certain amount of CPU time which it can schedule among its processes
    - 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling



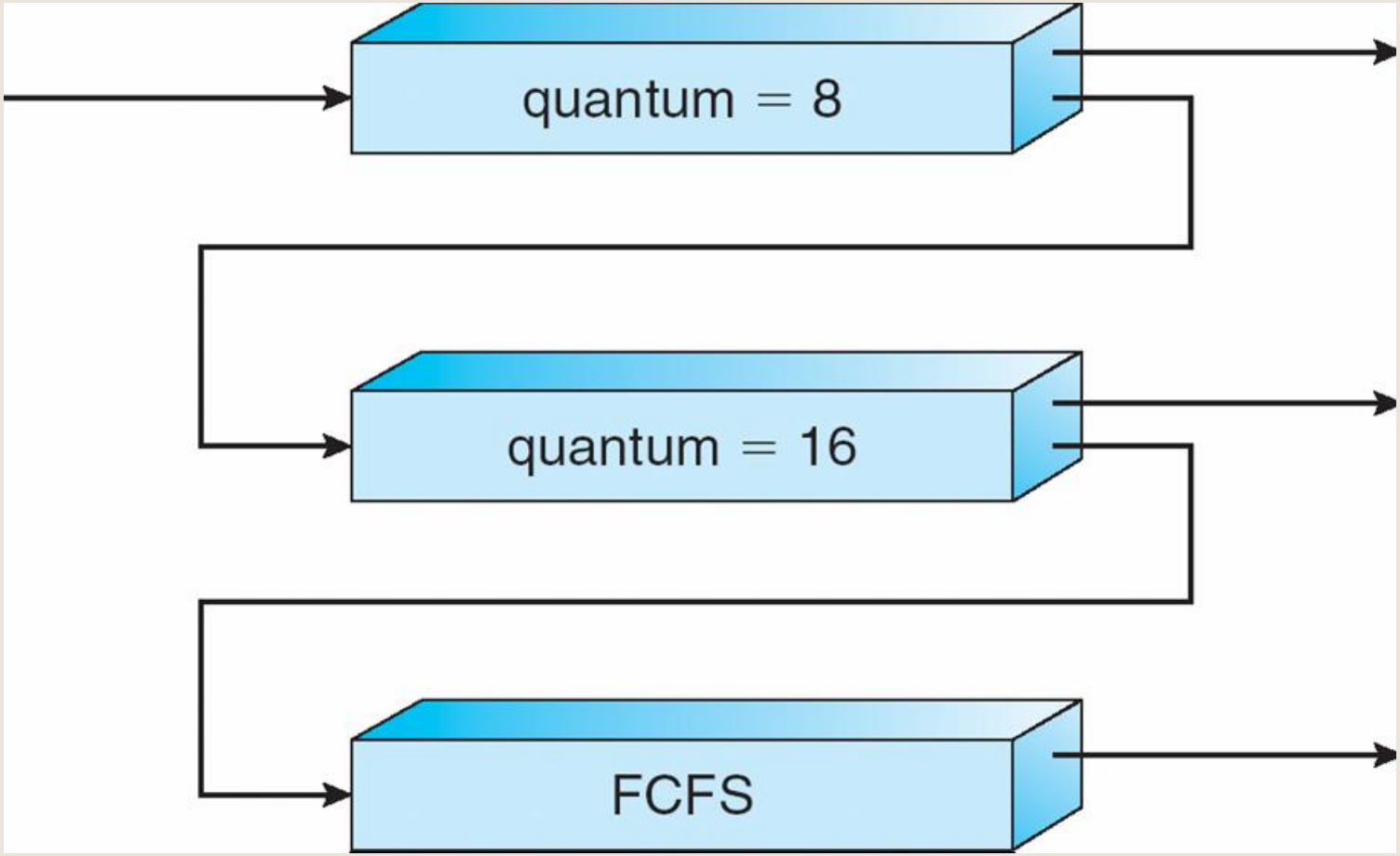
Multilevel Feedback Queues

- A process can move between the various queues
  - aging can be implemented in this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - methods used to determine when to upgrade a process
  - methods used to determine when to demote a process
  - methods used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters in the rear of the queue Q_0.
    - When a Q_0 job gains CPU, it receives 8 milliseconds.
    - If it does not finish in 8 milliseconds, the job is moved to the rear of the queue Q_1.
  - When a Q_1 job gains CPU, it receives 16 milliseconds.
    - If it still does not complete, it is preempted and moved to the rear of the queue Q_2.
  - When a Q_2 job gains CPU, it runs until it finishes.

Multilevel Feedback Queues



Real Time Applications

- Time critical applications usually require guaranteed response.
- A start time is the time at which a task must start in response to some event, such as an interrupt from a sensor.
- A deadline is the time at which the task must complete.
- A hard deadline is one in which there is no value if the deadline is missed.
- A soft deadline is one where there is decreasing value in the computation as the deadline slips.

Real-Time (RT) Systems

- A real-time (RT) system must be able to respond to events quickly.
  - Hard real-time systems can guarantee response times.
  - Soft real-time systems provide guaranteed responses at best effort.
- Real-time operating systems usually use priority schedulers.
  - All interrupt services must have a guaranteed maximum service time.
  - Efficient memory allocation, multi-threaded execution, and preemptable system calls are also supported.
  - RT processes must be fully loaded into the system's memory.
    - Otherwise, deadlines may be violated because the OS is busy doing something else.

RT Processes

- A tasks may be a terminating process, which run and then exit.
  - Its deadline is the latest time that it must exit.
- A task may also be nonterminating process.
  - It may run for a long time but perform periodic operations, such as encoding and decoding audio and video.
  - Its deadline is not the time to exit but rather the time by which results have to be ready for each period of execution.
    - e.g., decode a video frame every 33 ms (30 frames per second.)
  - Compute time is the time to perform each periodic event.
  - Deadline is the time that each periodic result must be ready.

If T = period, D = deadline, C = compute time,
$C \leq D \leq T$

RT Scheduling

- Earliest Deadline First (EDF) Scheduling:
  - Each RT process tells OS its deadline
  - Scheduler picks the process that has the nearest deadline to run.
    - In general, a process can run to completion if it has an earlier deadline.
    - However, it may be preempted if a process with an even earlier deadline starts.
- Least Slack First (LSF) Scheduling:
  - Consider both remaining compute time and deadline.
  - Look not only at the deadline but how much a process can be deferred.
    - slack = (time to deadline) – (amount of remaining computation)
    - E.g., suppose C = 5 ms, D = 20 ms from now, then slack = D - C = 15 ms

LSF vs. EDF

- Earliest Deadline First
  - The earliest deadline process always finish before others.
- Least Slack First can get a more fared (balanced) result.
- The differences to deadlines can be balanced If there's not enough time for everything:
  - In EDF, only some early deadline processes may finish.
  - In LSF, all deadlines may be missed but roughly by the same amount.

Rate Monotonic Scheduling

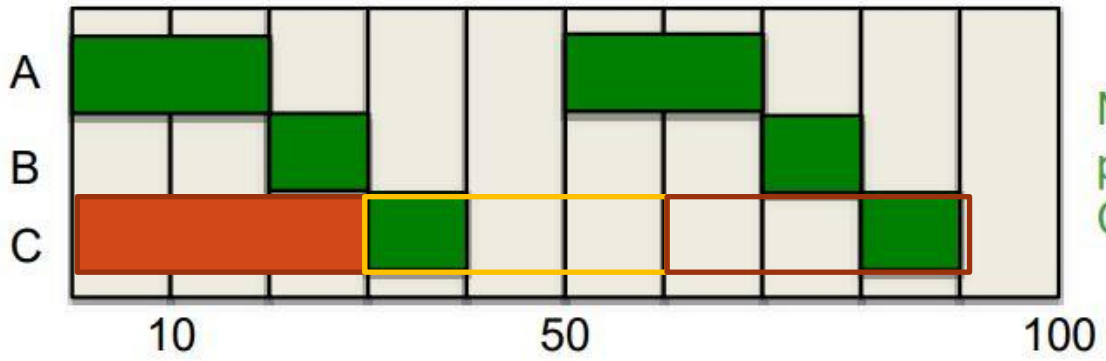
- Assign static priorities to periodic processes according to their periods.
  - Highest frequency (smallest period) process gets the highest priority
  - Lower frequency processes get lower priorities
  - All periodic RT processes must have known periods and are running at the same time.
- Scheduling periodic RT processes via a simple priority scheduler.
  - If two processes have the same priority, they can round robin.
- Rate monotonic scheduling is optimal.
  - If it is possible for all deadlines to be met then they will be met with rate monotonic scheduling.

Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the ACM*, **20** (1): 46–61

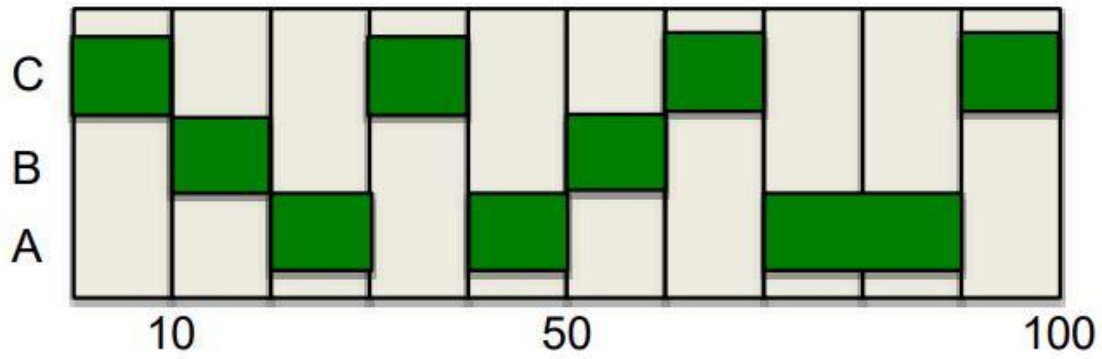
Rate monotonic example

- Process A runs every 50 ms for 20 ms
- Process B runs every 50 ms for 10 ms
- Process C runs every 30 ms for 10 ms

Rate monotonic analysis:
Schedule C first, then A or B



No rate monotonic priority assignment:
C misses a period!



CPU Memory Access

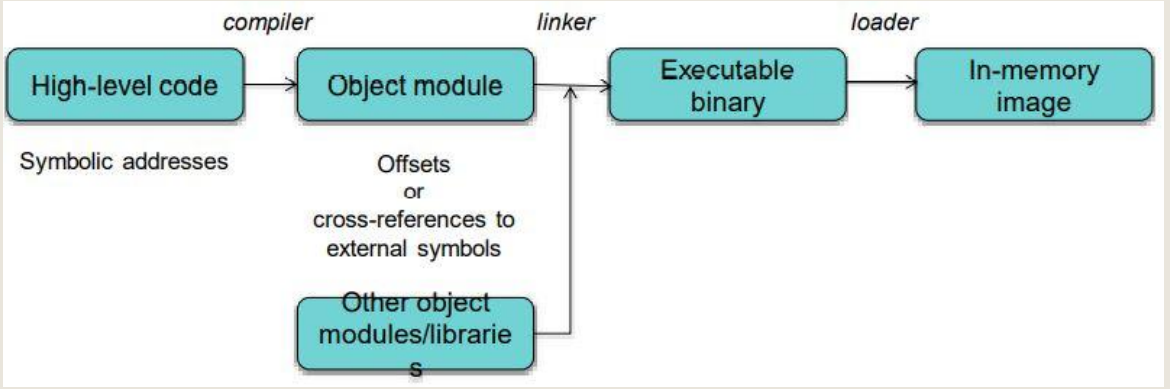
- A CPU executes a process by reading instructions.
  - These instructions reside in memory and they may request the processor to read and write other memory locations.
    - Each instruction and each datum is read from a memory location (an address.)
- Addresses may be either absolute or relative.
  - An absolute address is the actual memory location that is being referenced.
  - A relative address is expressed as an offset to the contents of a register.
  - The most common relative addressing mode is PC-relative, where the address is expressed as an offset to the program counter.

Binding of Instructions and Data to Memory

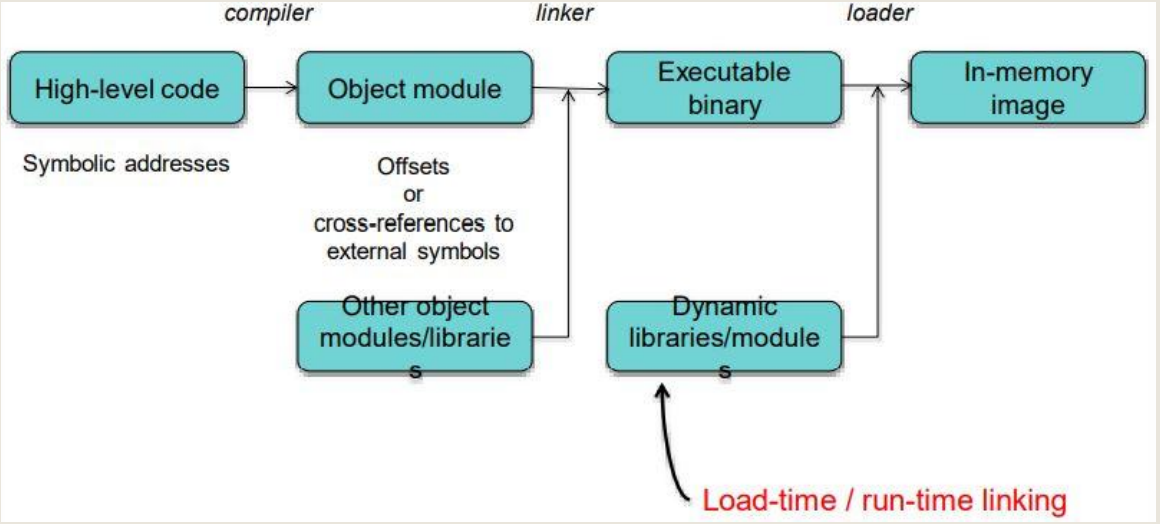
- Address binding of instructions and data to memory addresses can happen at three different stages.
  - Compile time: If memory location known a priori, absolute code can be generated.
    - Recompilation is needed if starting location changes.
  - Load time: Must generate relocatable code if memory 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)

Static & Dynamic Linker

Static Linker



Dynamic Linker



How do programs specify memory access?

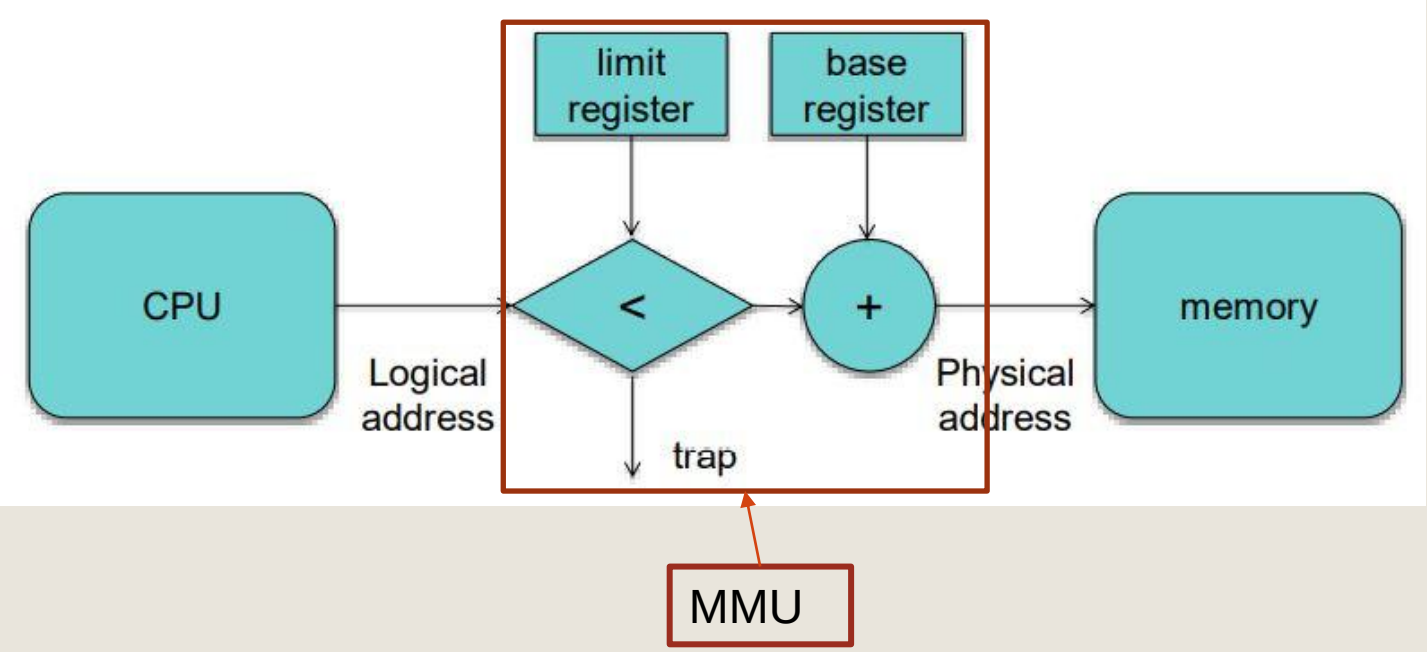
- Absolute code
  - If you know where the program gets loaded (any relocation is done at link time)
- Dynamically relocatable code
  - Relocated at load time
- Position independent code
  - All addresses are relative (e.g., gcc –fPIC option)
- Or … use logical addresses
  - Absolute code with addresses translated at run time
  - Need special memory translation hardware
    - E.g. memory management unit (MMU)

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 called virtual address.)
  - Physical address – address seen by the memory unit
- In compile-time and load-time address-binding schemes, logical and physical addresses are the same.
- In execution-time address-binding scheme, logical (virtual) and physical addresses can be different.
- In MMU scheme, 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 only it never sees the real physical addresses.

MMU Based Relocatable Addressing

- Base & limit registers inside MMU
  - Physical address = logical address + base register
  - But first check that: logical address < limit



Program Linking

- An executable program is typically built from several source files and makes references to library functions.
- A linker is responsible for combining these separately-compiled files and filling in undefined symbols in one module with their definitions from another module.
  - Static linking is the case where a linker resolves all symbols and includes all the code that is needed to run the program within the executable file.
    - The operating system's program loader can load it into memory and run it.
  - Dynamic linking can leave certain libraries and symbols unresolved but are loaded on demand until they are referenced in runtime.
    - The dynamic linker usually links the program with stub libraries whose only task is to load the actual libraries.

Dynamic Loading

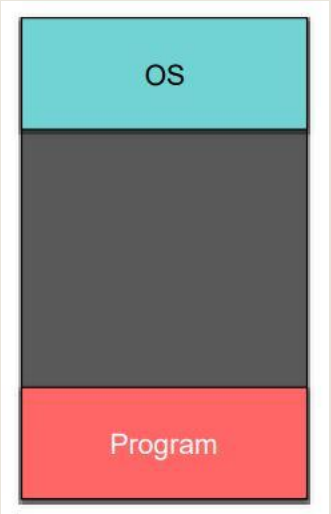
- A subroutine is not loaded until it is called
- Better memory-space utilization
  - Unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - It can be implemented through proper program design

Shared Library

- Dynamic linking enables the use of shared libraries.
  - The same block of code containing the library can be referenced by multiple processes that need it.
  - If a unloaded shared library is referenced, the program loader will allocate memory and load it from the library file.
  - If a loaded library is referenced, the program loader will replace the process' references to the memory that library was loaded.
- Since a shared library may be loaded in different memory location at each memory residency, it must be compiled to use relative addressing (also called position independent code).
  - The OS keeps track of the number of references to each shared library.
    - When the last process that uses a shared library terminates, the library is then unloaded from memory.

Memory Management

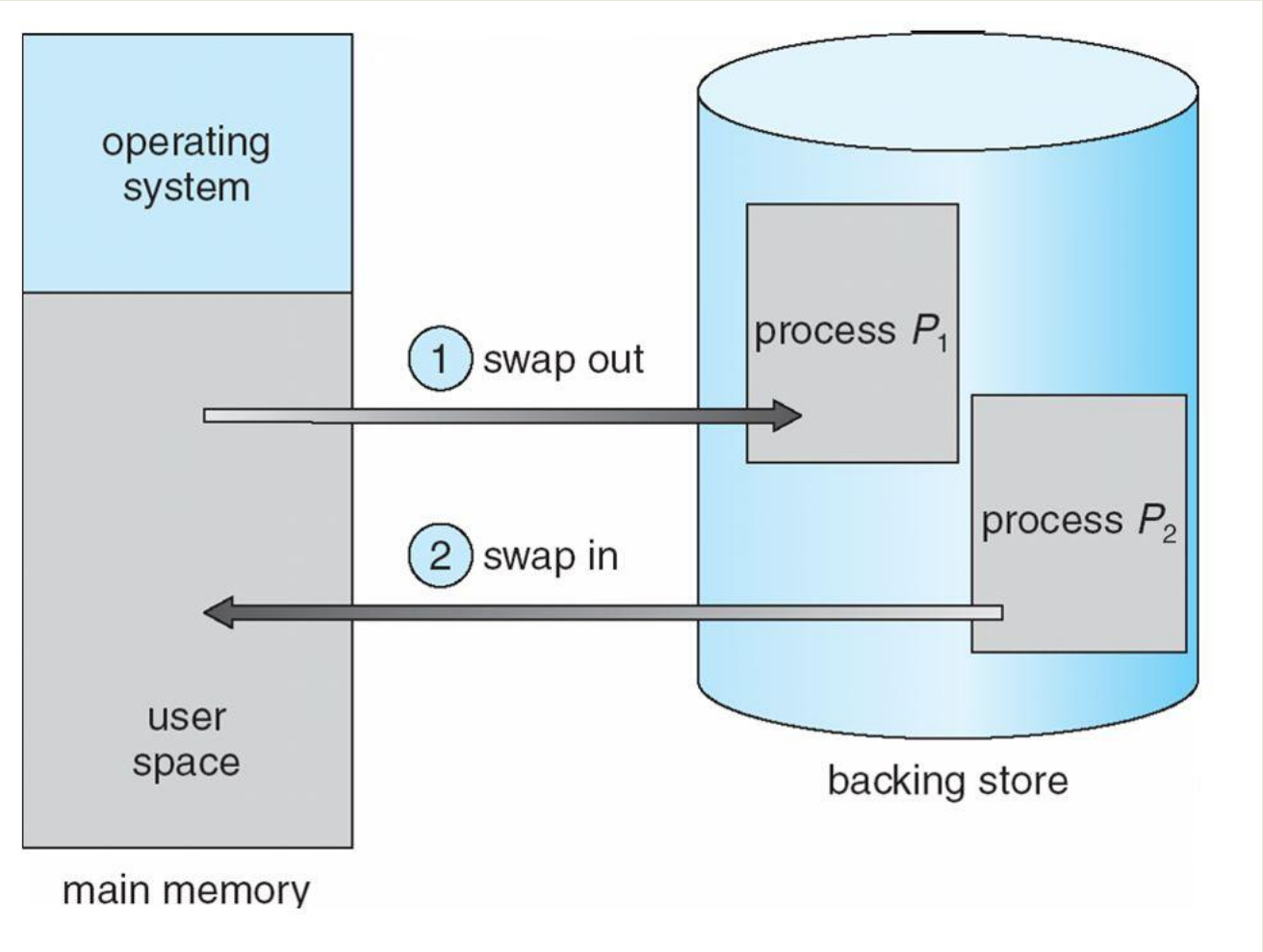
- The simplest approach is the single partition multiprogramming.
  - Only a single program is loaded in memory at any given time.
  - The program shares memory with the OS and nothing else.
  - Absolute addressing is no problem.
  - Many older systems (e.g. MSDOS) use this approach.



Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- A backing store can be a fast disk large enough to accommodate copies of all memory images for all users.
  - It must provide direct access to these memory images.
- Roll out, roll in is a swapping variant used for priority-based scheduling.
  - Lower-priority processes are swapped out so higher-priority processes can be loaded and executed.
- Major part of swap time is the total transfer time that is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems.
  - i.e., UNIX, Linux, and MS Windows
- OS maintains a ready queue of ready-to-run processes which have memory images on disk.

Schematic View of Swapping

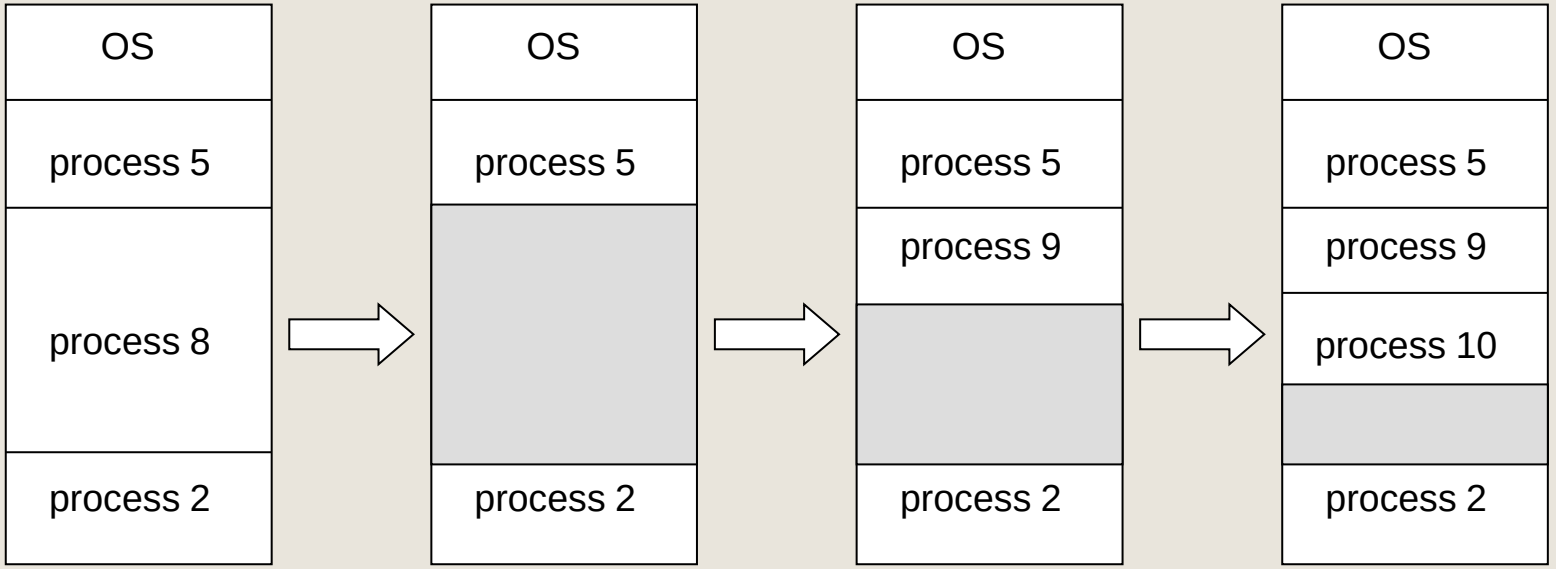


Multiple Fixed Partitions

- Divide memory into predefined partitions (segments)
  - Partitions don't have to be the same size
  - For example: a few big partitions and many small ones
- New process gets queued for a partition that can hold it
- Unused memory in a partition is wasted: internal fragmentation
- Unused partitions are wasted also: external fragmentation
- Process takes up a contiguous region of memory (Contiguous allocation.)
  - Relocation registers (MMU) are used to protect user processes from each other and from changing operating-system code and data.
    - Base register contains value of smallest physical address.
    - Limit register contains range of logical addresses.
  - Every logical address must be less than the value in limit register .

Multiple Variable Partitions (MVP)

- A Hole is a block of available memory.
  - Holes of various size are scattered throughout memory.
- When a process arrives, it is allocated to a large enough hole to accommodate it.
- OS keeps track of all allocated partitions and free partitions (holes.)



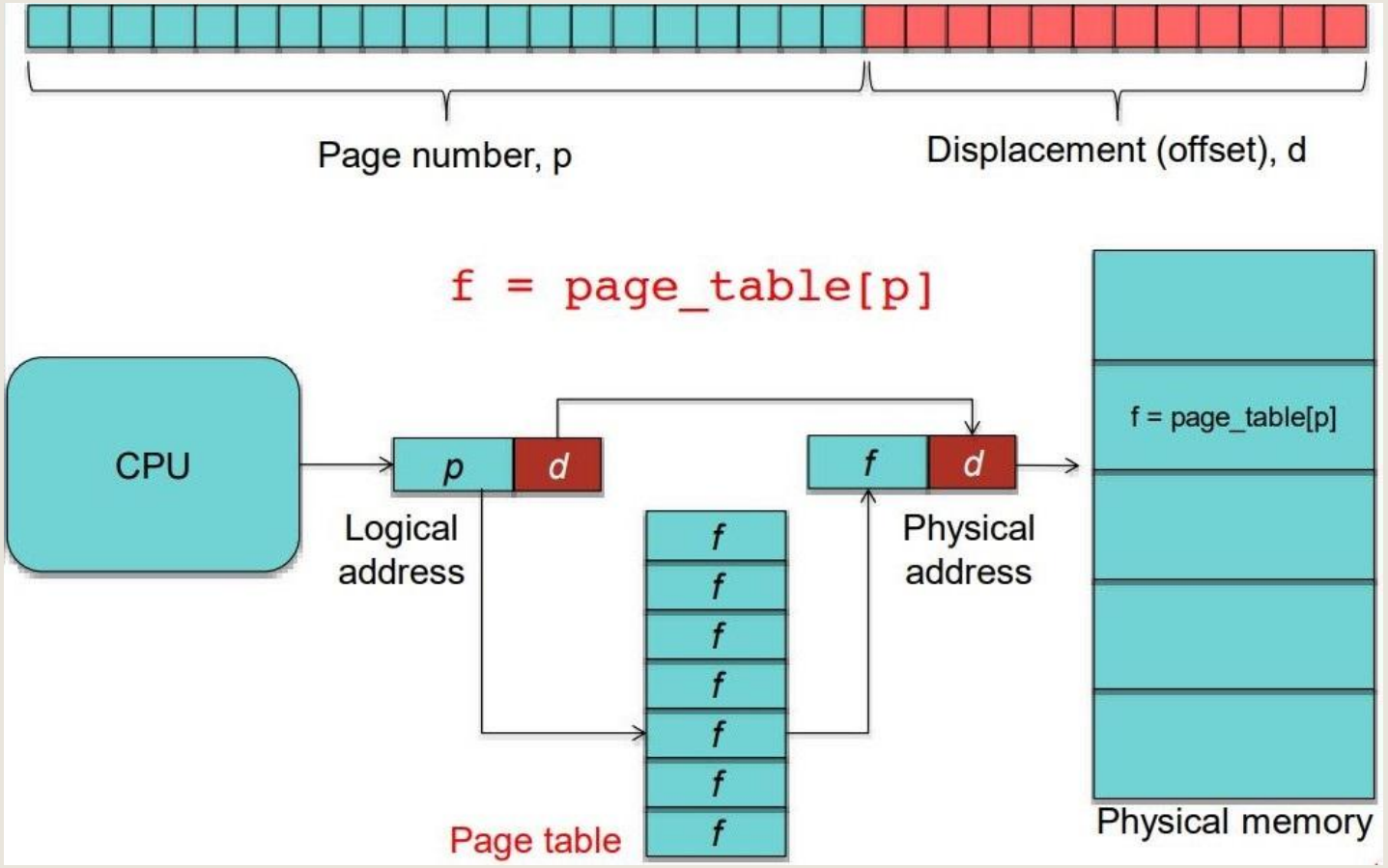
Allocation Algorithms for MVP

- How to satisfy a request of size n from a list of free holes.
  - First-fit: allocates the first hole that is big enough.
  - Best-fit: allocates the smallest hole that is big enough.
    - It must search entire list, unless holes are ordered by size.
    - It produces smallest and maybe useless (external fragmentation) leftover holes.
    - To avoid creating a very small useless hole, the smallest big enough hole may be assigned to the new process as a whole (internal fragmentation).
  - Worst-fit: allocates the largest hole
    - It must also search entire list, unless holes are ordered in descending order.
    - It always produces the largest leftover hole.
- In general, first-fit and best-fit perform better than worst-fit in terms of speed and storage utilization

Page-Based Memory Management

- Logical address space of a process can be noncontiguous.
- Physical memory is divided into fixed-sized blocks called frames.
  - In general, frame size is power of 2 (512 bytes ~8,192 bytes.)
- Logical memory of each process is also divided into blocks of same size called pages.
- OS must keep track of all free frames.
  - To run a program of n pages, n free frames must be located.
- Page table is used to translate logical to physical addresses.
- External fragmentation can be eliminated.
- Internal fragmentations may exist.
  - The last page is usually not a full page.
  - In average, each process has a half page internal fragmentation.

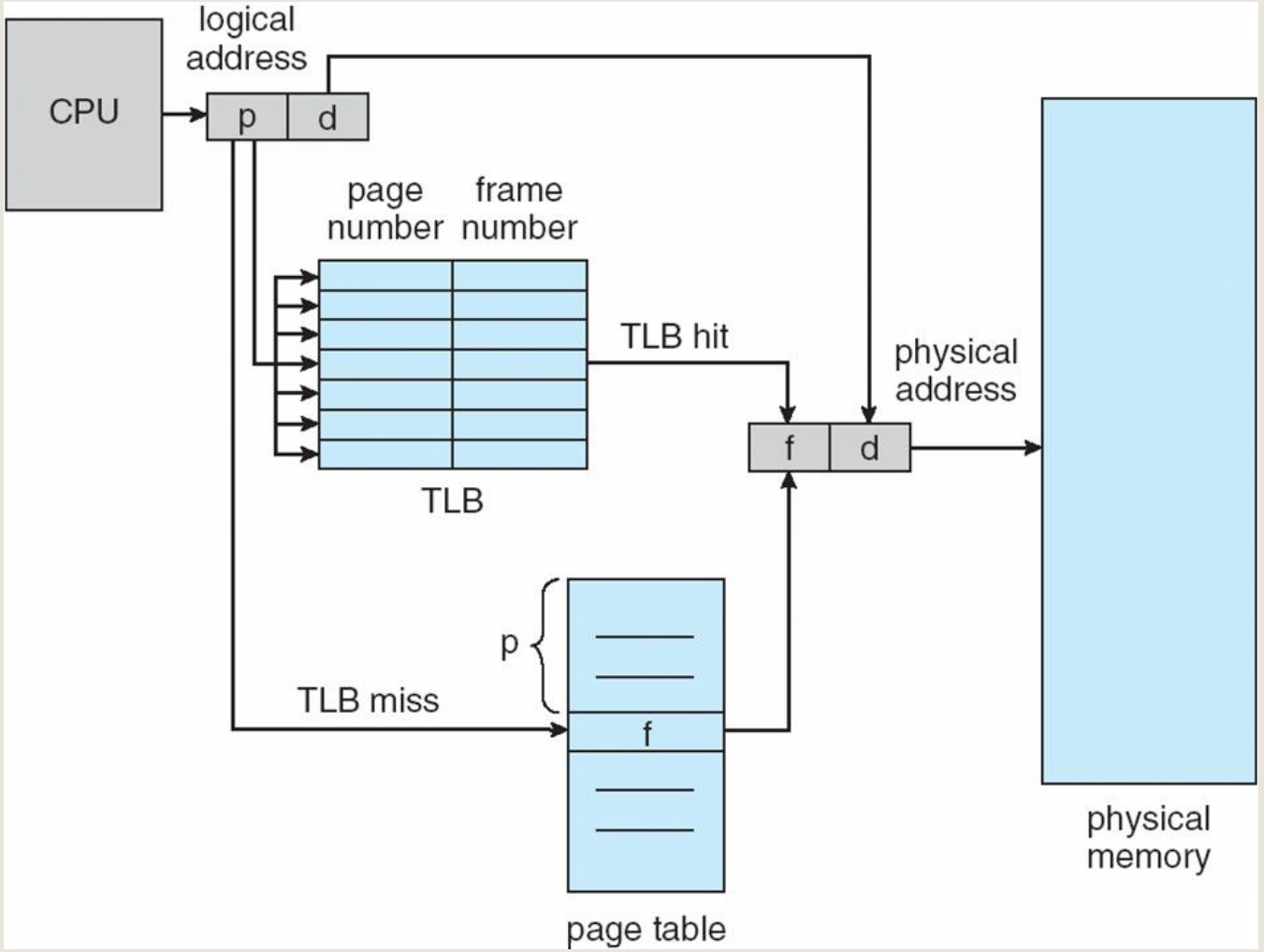
Page Translation



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 (PRLR) indicates size of the page table
  - 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 associative memory or translation look-aside buffers (TLBs.)
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry.
  - This can uniquely identify each process to provide address space protection for that process.

Paging With TLB



Effective Access Time with TLB

- Let the TLB access time is 10^{-6} sec, the main memory access time is 10^{-3} sec, and TLB hit ration is 90%
- A memory access may lead to
  - If TLB hit, 1 TLB search + 1 memory access
  - Otherwise, 1 TLB search + 2 memory access
- Effective memory access time becomes

  $\Rightarrow 0.9*(10^{-6}+10^{-3})+0.1*(10^{-6}+2*10^{-3})$

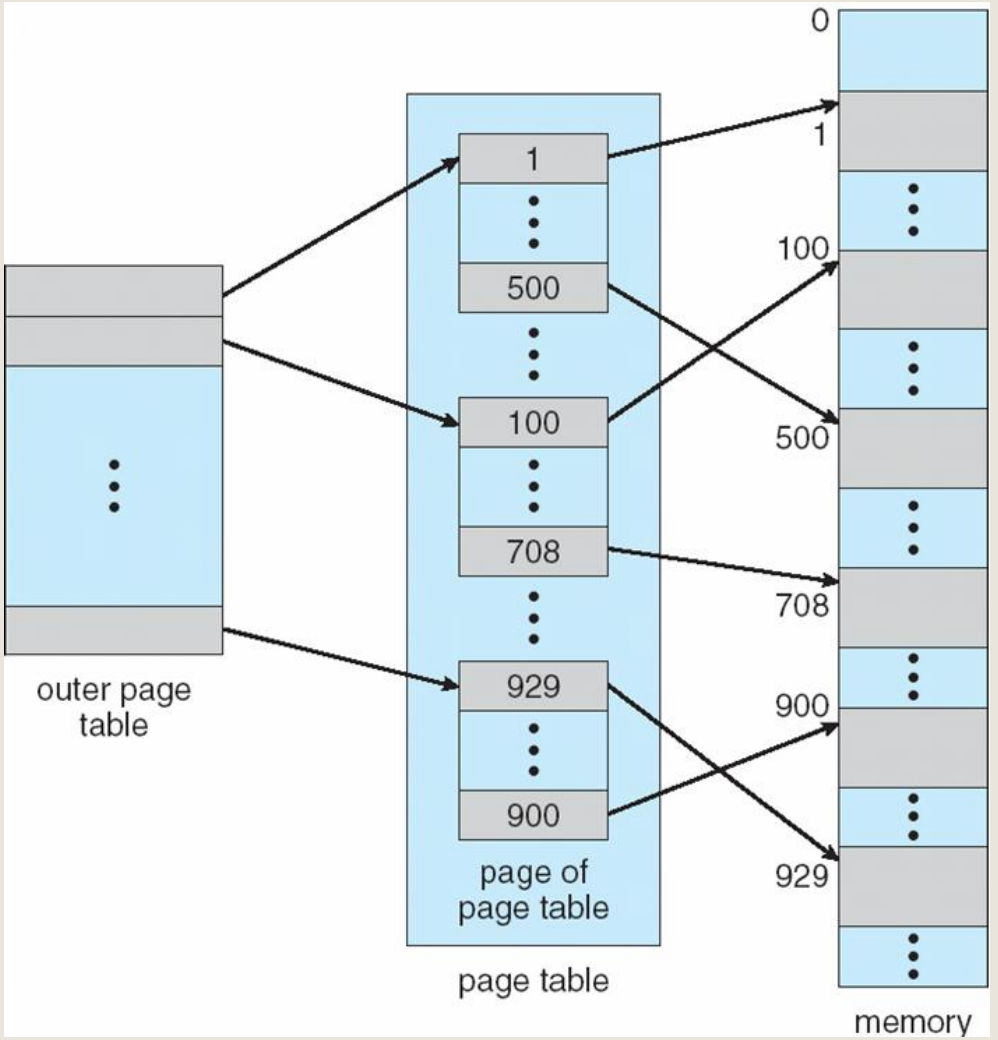
  $\Rightarrow 10^{-6}+1.1*10^{-3} \approx 1.1$ of the memory access time

Paging for Large Address Space

- Most processes use only a small part of their address space.
  - Keeping an entire page table is wasteful.
    - Example: 32-bit system with 4KB pages:
      - 20-bit page table $\Rightarrow 2^{20} = 1{,}048{,}576$ entries in the page table
- Three common approaches to deal with large address space are:
  - Hierarchical Paging uses multilevel page tables
  - Hashed Page Tables
  - Inverted Page Tables

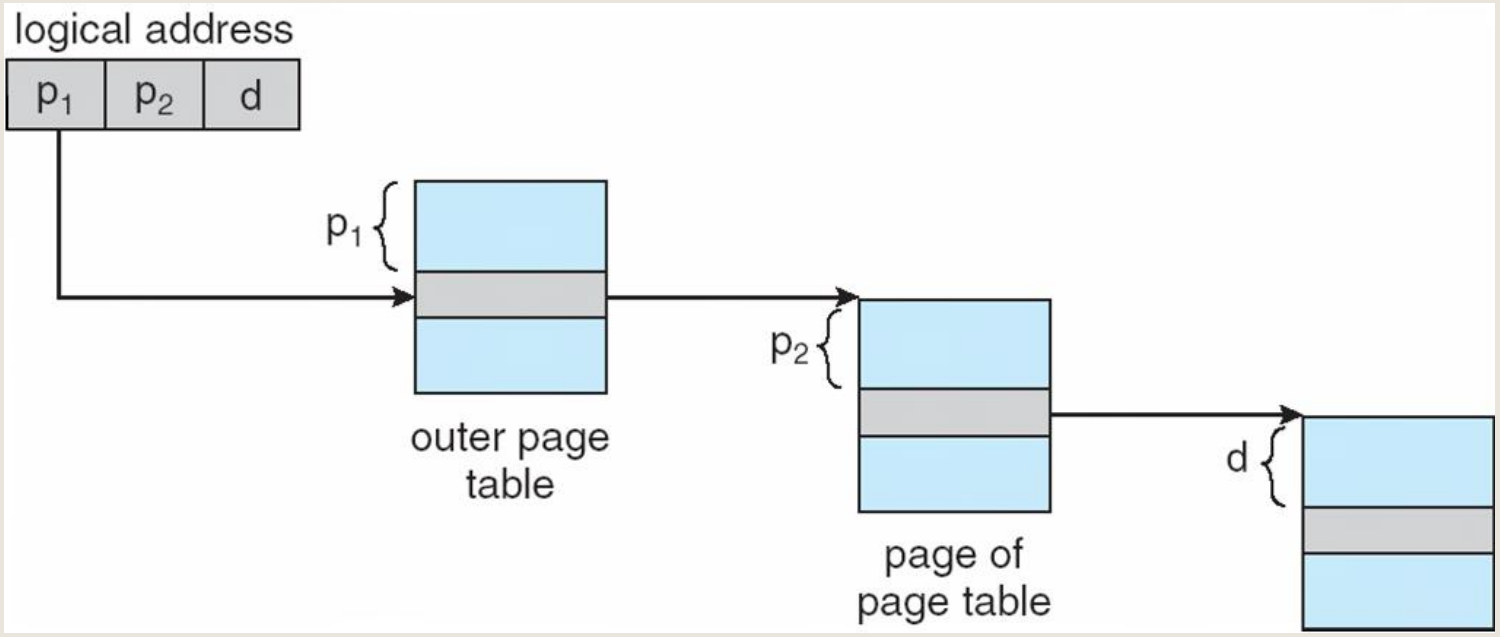
Hierarchical Paging

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
  - A 32-bit logical address of 1K page size is divided into:
    - a 22-bit page number
    - a 10-bit page offset
  - The page number is then paging into:
    - a 12-bit outer page number
    - a 10-bit inner page number



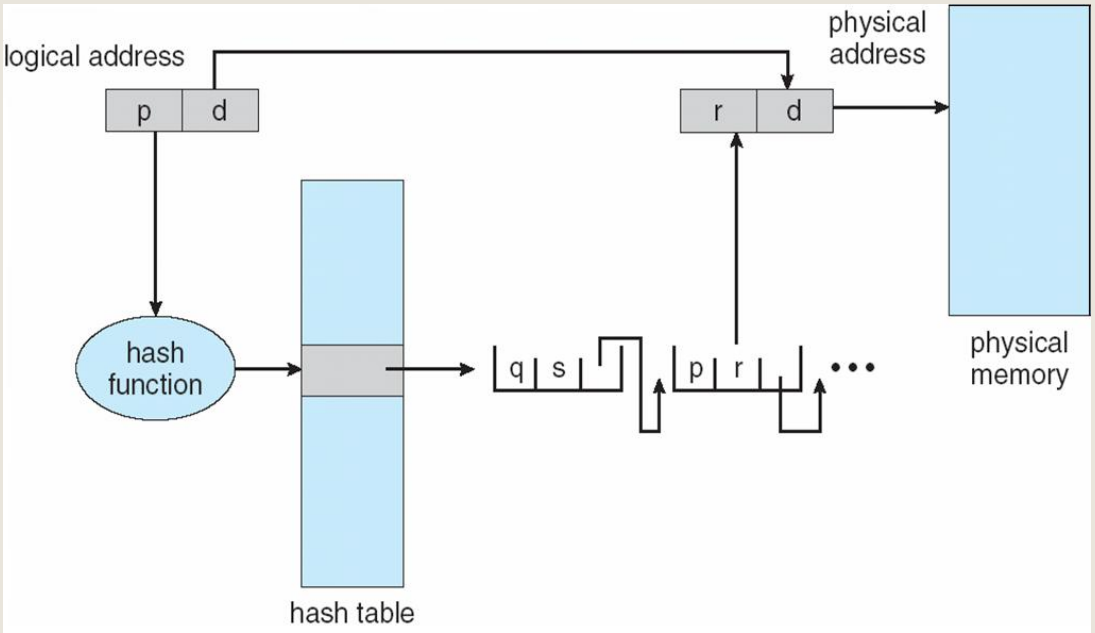
Address-Translation in Two-Level Paging Scheme

page number		page offset
p_1	p_2	d
12	10	10



Hashed Page Tables

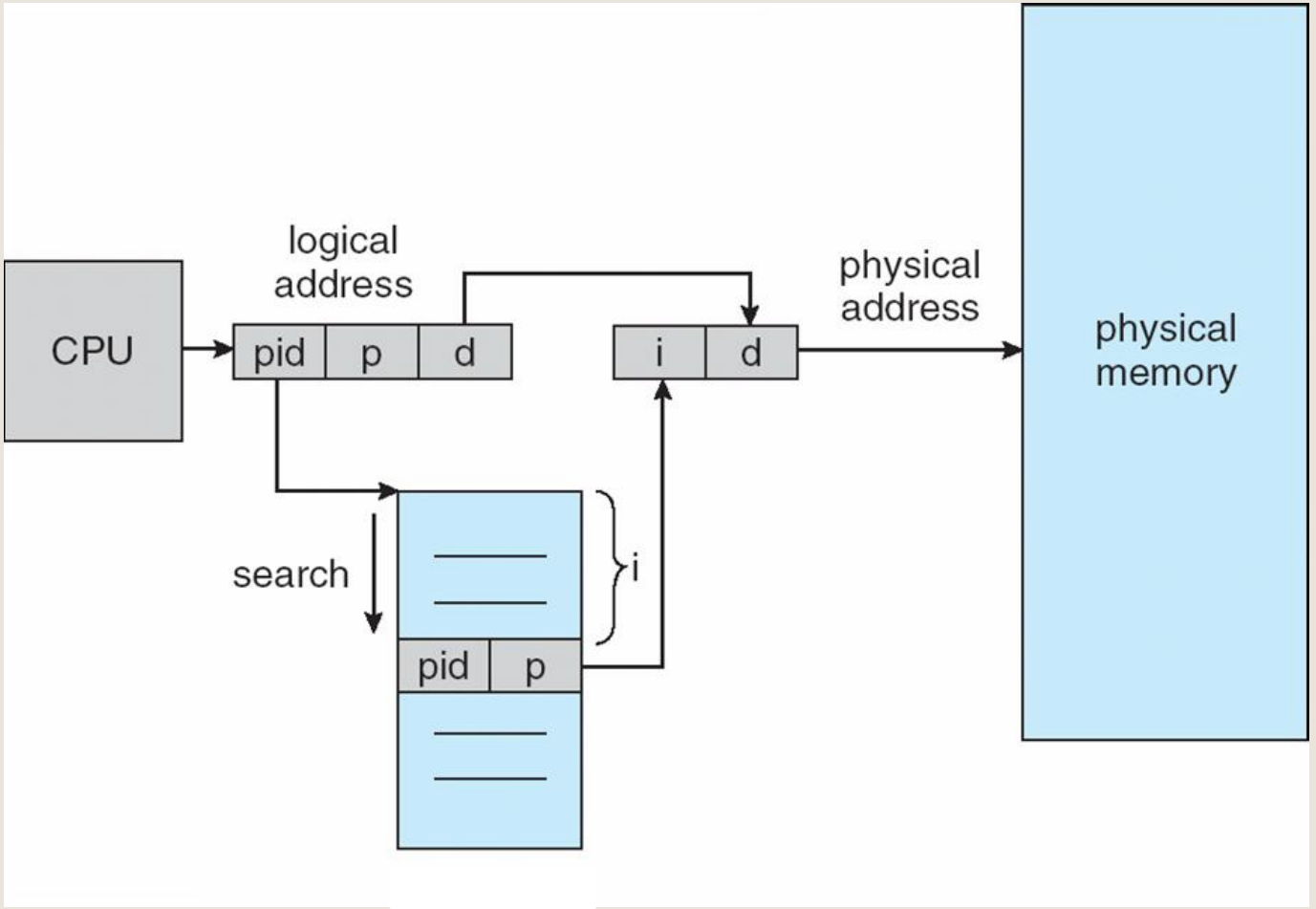
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table.
  - This page table contains a chain of elements hashing to the same location.
  - Virtual page numbers are compared in this chain searching for a match.
    - If a match is found, the corresponding physical frame is extracted.



Inverted Page Table

- Used when the # of frames is more manageable than the # of pages.
- The i-th entry contains which page is in frame i.
  - It also includes information about the process that owns that page.
- It can reduce memory needed to store page tables, but increases time needed to search the table when a page reference occurs.
  - Hash table can be used to limit the search to one or at most a few page table entries.

Inverted Page Table Architecture



Inverted page table