Operating System Concepts

Lecture 23: Memory Management

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- diving deeper into the concepts and mechanisms of memory sharing and address translation
  - how memory addresses generated by CPU are mapped to physical addresses?
  - what's the difference between static and dynamic relocation?
  - what are the policies for contiguous memory allocation?
  - what is internal/external fragmentation and how to eliminate it?

Closer look at the instruction execution cycle

- instruction execution cycle
  - **step 1**: fetch instruction from memory according to the value stored in the program counter
  - **step 2**: decode the instruction; this may cause operands to be fetched from memory
  - **step 3**: execute the instruction
  - **step 4**: write the result back to memory, or store it in a register

Closer look at the instruction execution cycle

- instruction execution cycle
  - **step 1**: fetch instruction from memory according to the value stored in the program counter
  - **step 2**: decode the instruction; this may cause operands to be fetched from memory
  - **step 3**: execute the instruction
  - **step 4**: write the result back to memory, or store it in a register
- CPU can directly access main memory and registers built into each core, for example: memory direct to register: **`MOV r1, @0xfffa620e`**
  - instructions cannot take disk addresses; hence, the program's code and data must be loaded into memory before the CPU can operate on them
  - registers are accessible within one cycle, but memory access can take many cycles of the CPU clock (CPU needs to **stall** in such cases)
  - cache is used for faster access

Basics of memory addressing

- a k-bit address allows referencing 2^k locations
  - each location can be one byte, 2 bytes, or 4 bytes (one word)
  - 2^{32} = ~4 billion addresses on a 32-bit machine
- each program operates in an address space distinct from the physical memory space of the machine

Basics of memory addressing

- a k-bit address allows referencing 2^k locations
  - each location can be one byte, 2 bytes, or 4 bytes (one word)
  - 2^{32} = ~4 billion addresses on a 32-bit machine
- each program operates in an address space distinct from the physical memory space of the machine
- protection: prevent access to private memory of other processes
- translation: map accesses from one address space (virtual) to a different one (physical)

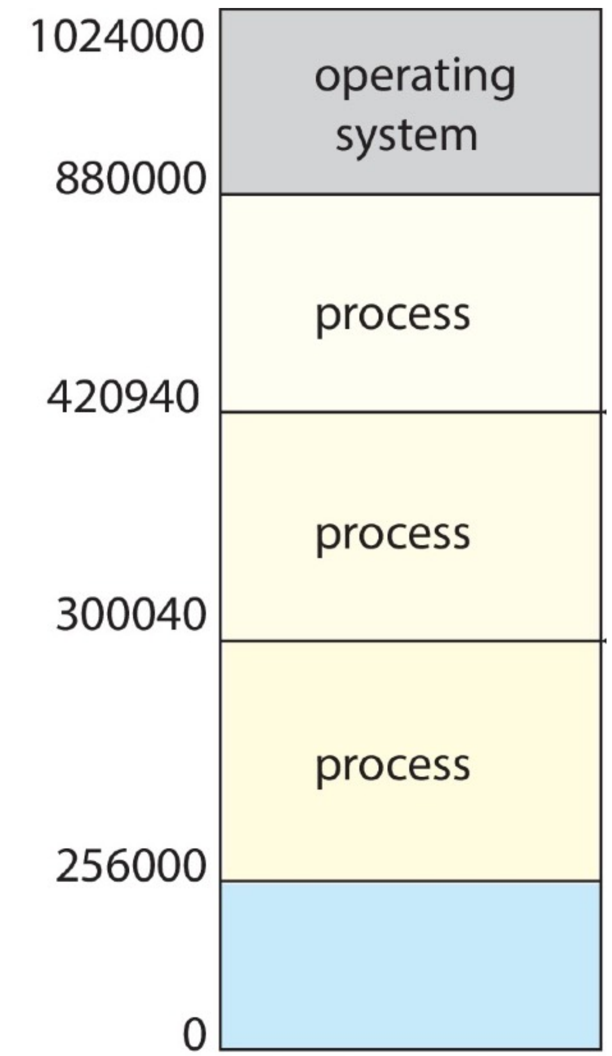
Uniprogramming

- works in simple systems, e.g., embedded computers, micro-controllers
- one program shares memory with the OS; it always runs at the same place in physical memory (no translation)
    - load application into low memory and OS into high memory
    - application can address <u>any physical memory location</u>
        - they can corrupt OS and even the disk (no protection)

Multiprogramming

multiprogramming without translation or protection (e.g., Windows 95)

- need to load multiple processes in memory for concurrent execution
  - they are unaware of sharing the memory with each other
  - translation is done by linker/loader; adjusts addresses while program loaded into memory
- compiler generates .o file with code starting at location 0
- linker scans through each .o file, changing addresses to point to where each module goes in the larger program
- loader loads the executable (a.out) to the memory and runs the program
- problem with just using linker-loader? still no protection
  - bugs in any program can cause other programs (even OS) to crash



Address translation terminology

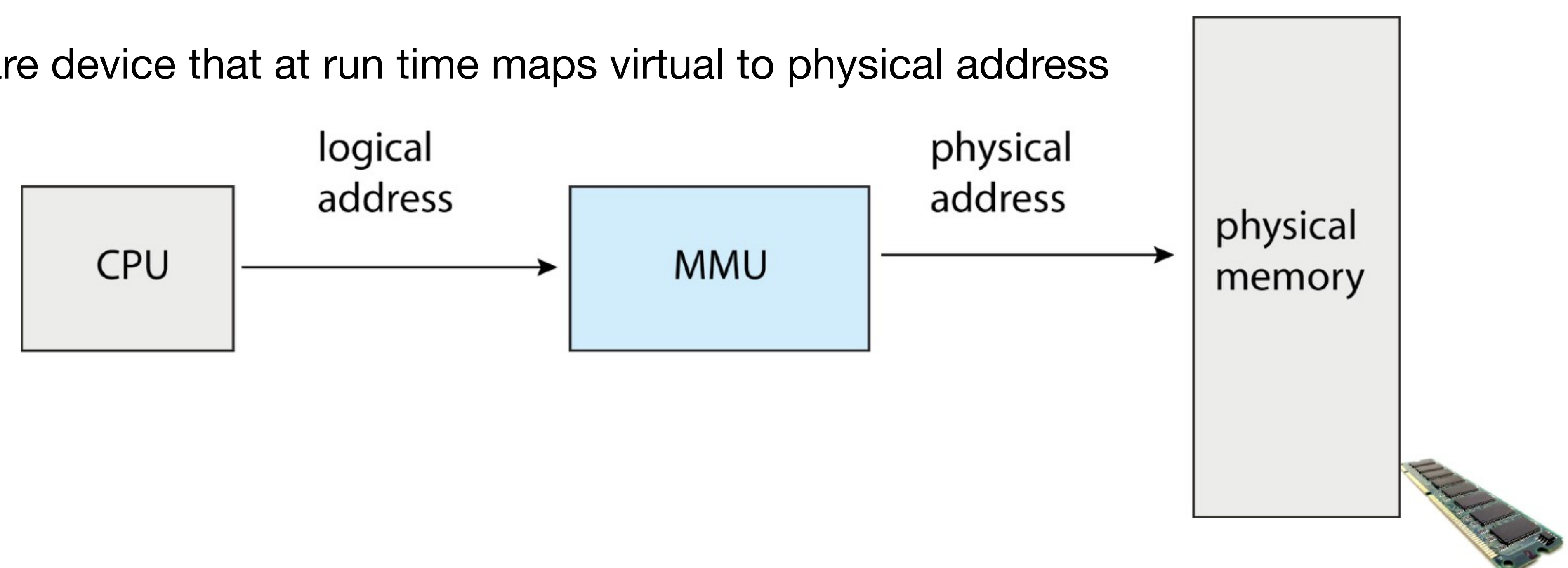
- logical/virtual address: address generated by the CPU/ issued by the program
  - starting from 0, going up to $MAX_{procddr}$

Address translation terminology

- logical/virtual address: address generated by the CPU/ issued by the program
  - starting from 0, going up to $MAX_{procddr}$
- physical address: address seen by the memory unit
  - starting from 0, going up to $MAX_{sysaddr}$
  - the physical address is seen by the memory unit and is loaded into the memory-address register of the memory

Address translation terminology

- logical/virtual address: address generated by the CPU/ issued by the program
  - starting from 0, going up to $MAX_{procddr}$
- physical address: address seen by the memory unit
  - starting from 0, going up to $MAX_{sysaddr}$
  - the physical address is seen by the memory unit and is loaded into the memory-address register of the memory
- memory management unit (MMU)
  - hardware device that at run time maps virtual to physical address



Address binding schemes

- can happen at three different stages
  - **compile time**: if memory location known a priori, absolute code can be generated; code must be recompiled if starting location changes
    - logical and physical addresses are the same
  - **load time**: must generate relocatable code if memory location is not known at compile time
    - logical and physical addresses can be the same
  - **execution time**: binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - needs hardware support for address mapping (e.g., base and limit registers)

Multiprogramming with protection

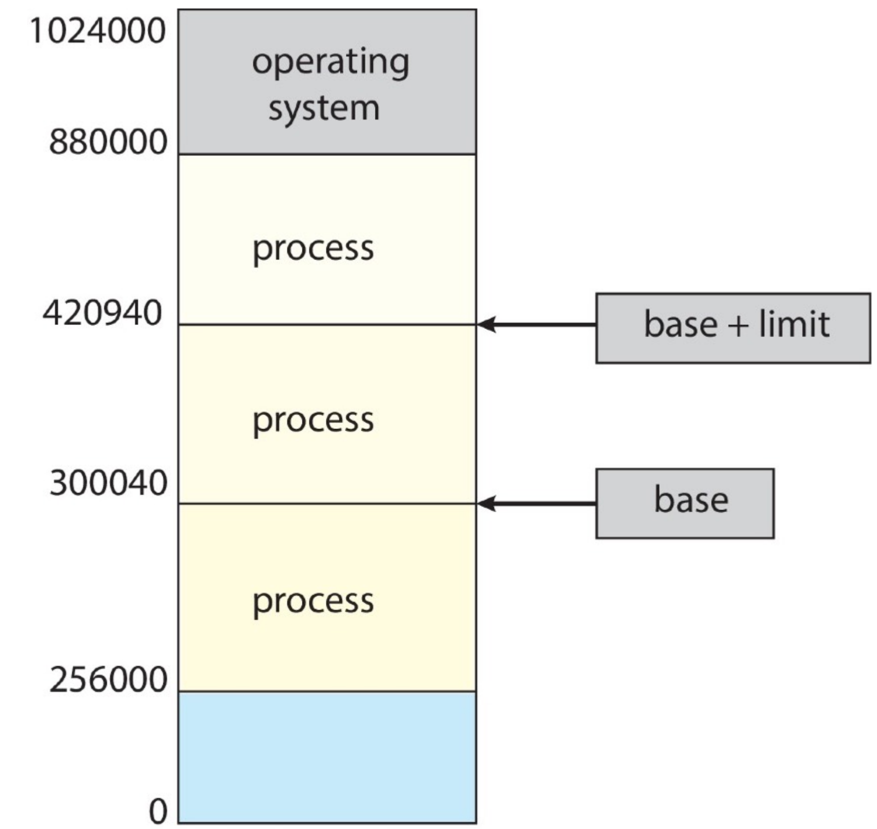
- protection is necessary
  - separate per-process memory space
  - processes don't care what physical portion of memory they are assigned to
- there are different options:
  - base and bounds
  - segmentation
  - paging
  - multi-level translation

Hardware support

- hardware cost: two registers and address comparators
  - base register holds the smallest legal physical memory address
  - limit register stores the size of the range

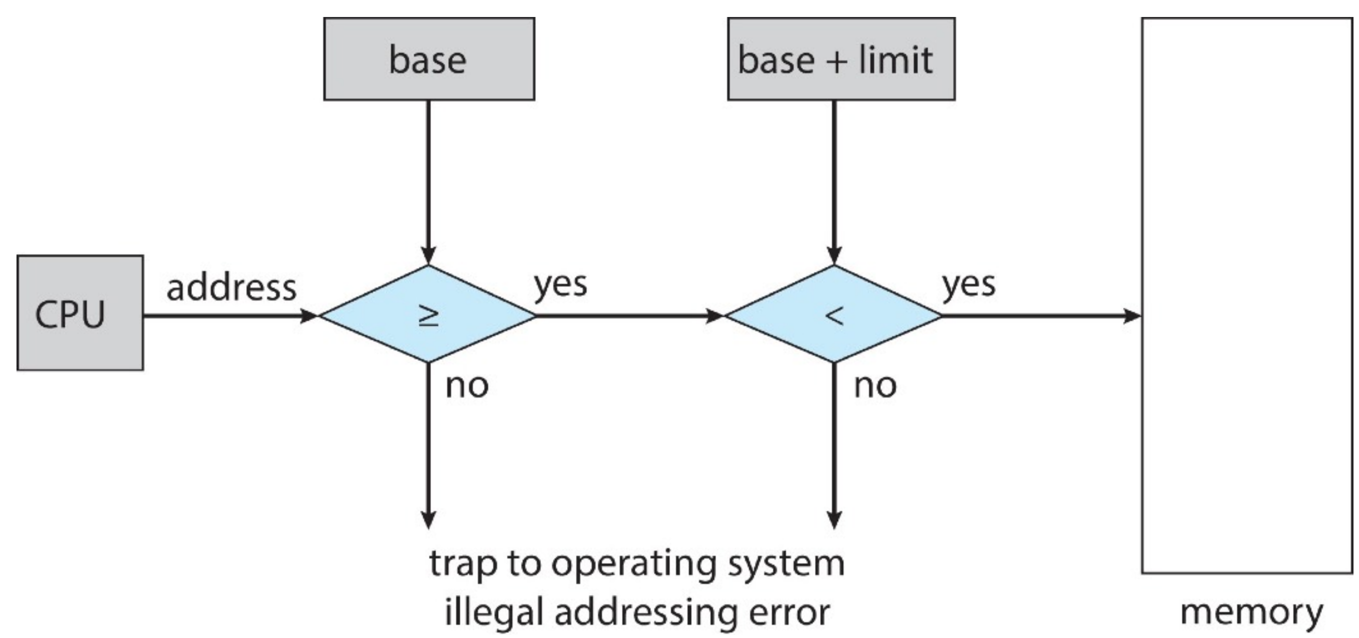
Hardware support

- hardware cost: two registers and address comparators
  - base register holds the smallest legal physical memory address
  - limit register stores the size of the range
- save and restore registers on a context switch
  - loading values into registers requires using a privileged instruction (in the kernel mode), hence it is done by the OS



Hardware support

- hardware cost: two registers and address comparators
  - base register holds the smallest legal physical memory address
  - limit register stores the size of the range
- save and restore registers on a context switch
  - loading values into registers requires using a privileged instruction (in the kernel mode), hence it is done by the OS
- an attempt to access an address outside this range results in a trap to the OS



Relocation

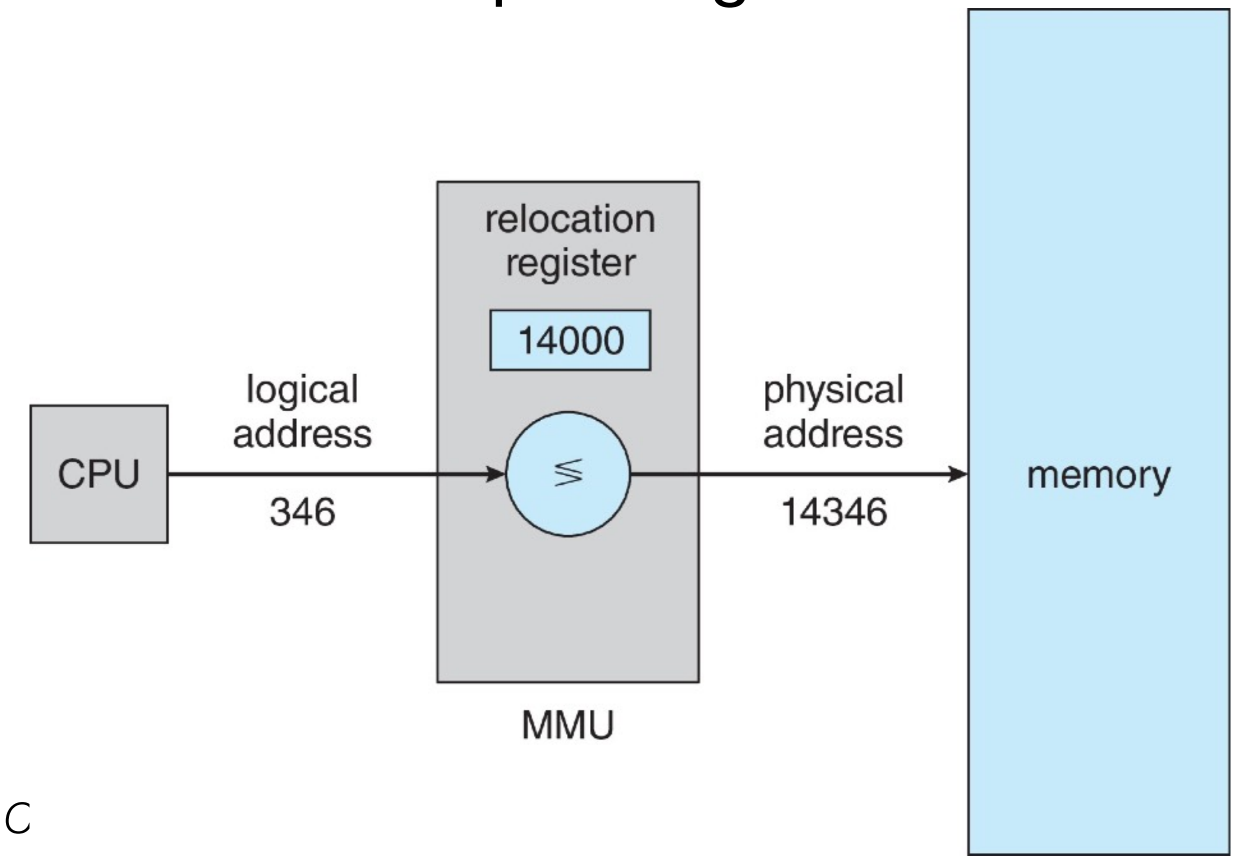
- program contains virtual addresses but the machine understands physical addresses
- if physical address==virtual address then
  - we cannot have multiple programs residing in memory at once
- hence, virtual addresses must be relocated to physical addresses at run time

Static relocation

- at load time, the OS adjusts the addresses in a process to reflect its position in memory
- once a process is assigned a place in memory and starts executing it, the OS cannot move it

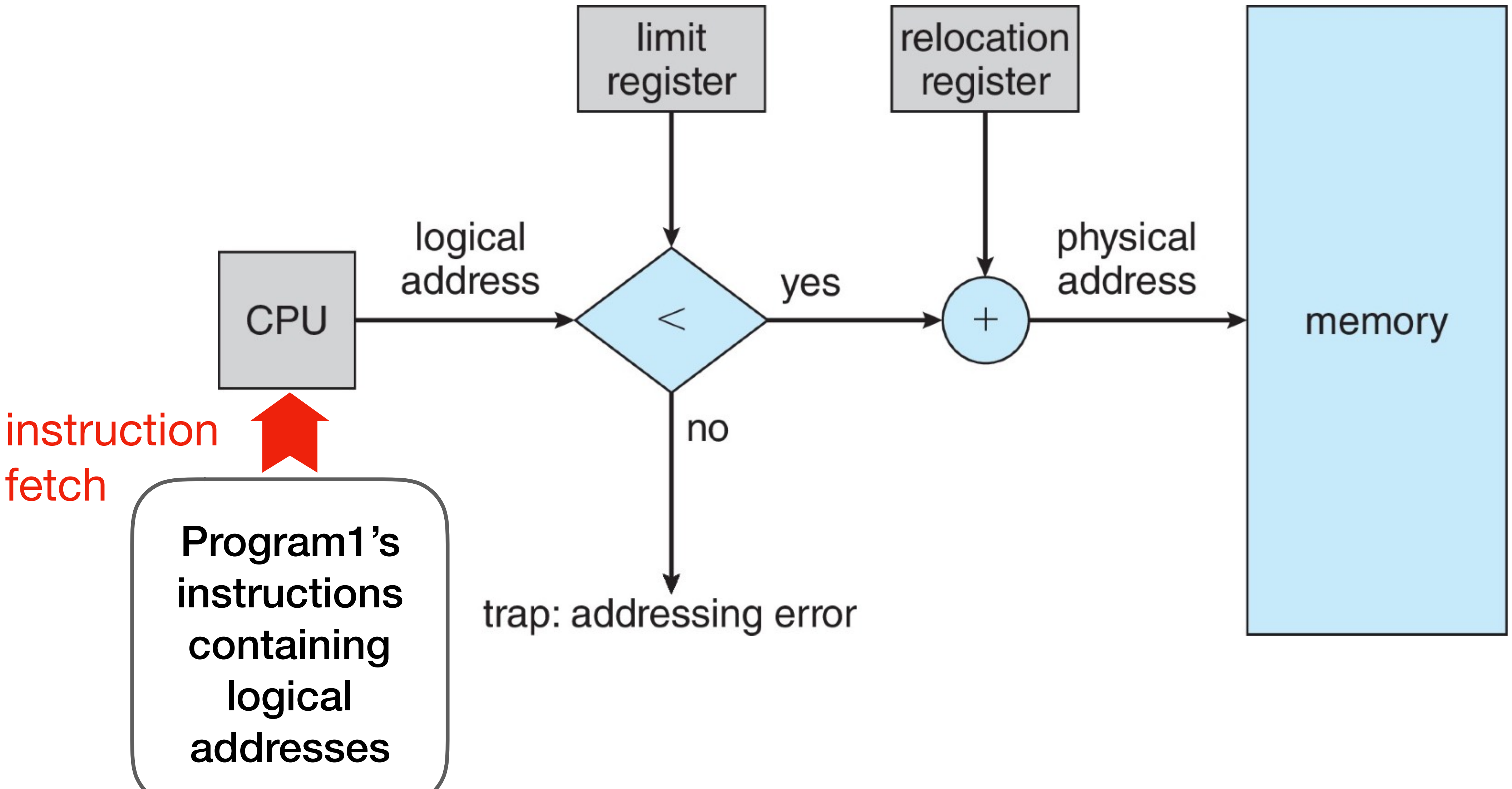
Dynamic relocation

- hardware adds the content of the **relocation register** (base register) to virtual addresses to get the corresponding physical addresses
  - addresses within program do not have to be relocated when program placed in different region of RAM
- hardware compares address with limit register (address must be less than limit)
- if test fails, the processor takes an address trap and ignores the physical address



Protection

- each memory reference is checked
- the base register is called the relocation register



Evaluating dynamic relocation

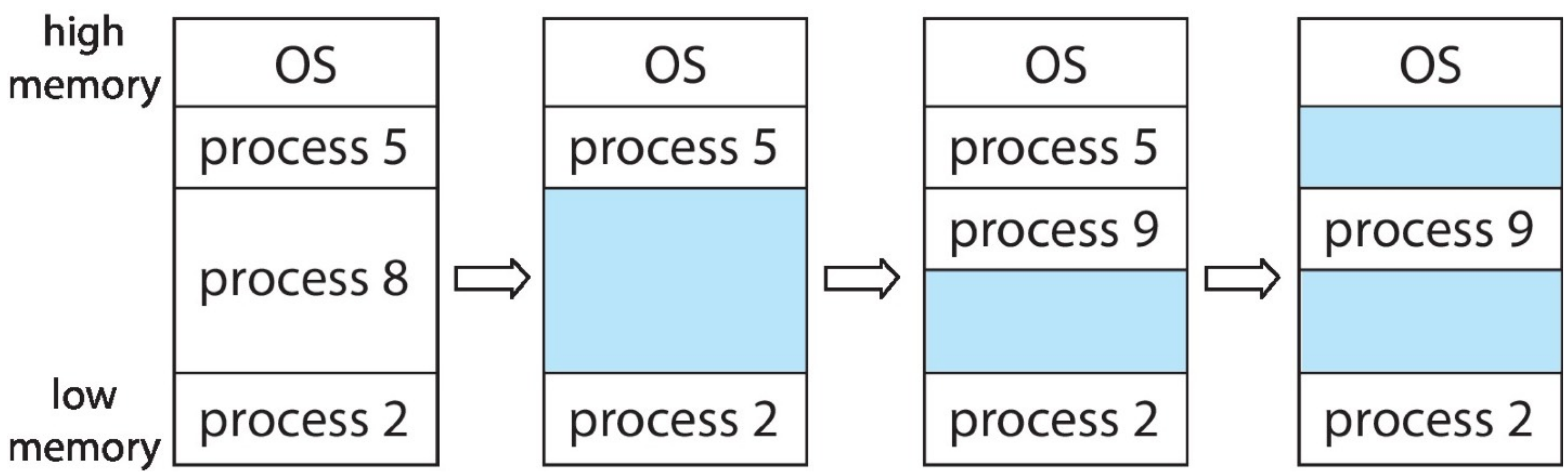
- advantages
  - OS can easily move a process in memory during its execution
  - simple: requiring two special registers, an add and a comparison
  - OS can allow a process to grow over time (it can be slow)

Evaluating dynamic relocation

- advantages
  - OS can easily move a process in memory during its execution
  - simple: requiring two special registers, an add and a comparison
  - OS can allow a process to grow over time (it can be slow)
- disadvantages
  - slow: every memory reference requires adding the content of the relocation register to the logical address
  - doesn't allow code sharing between processing
  - all memory of an active process must fit in the memory, reducing the degree of multiprogramming
  - the size of the memory image of a process is limited to physical memory size

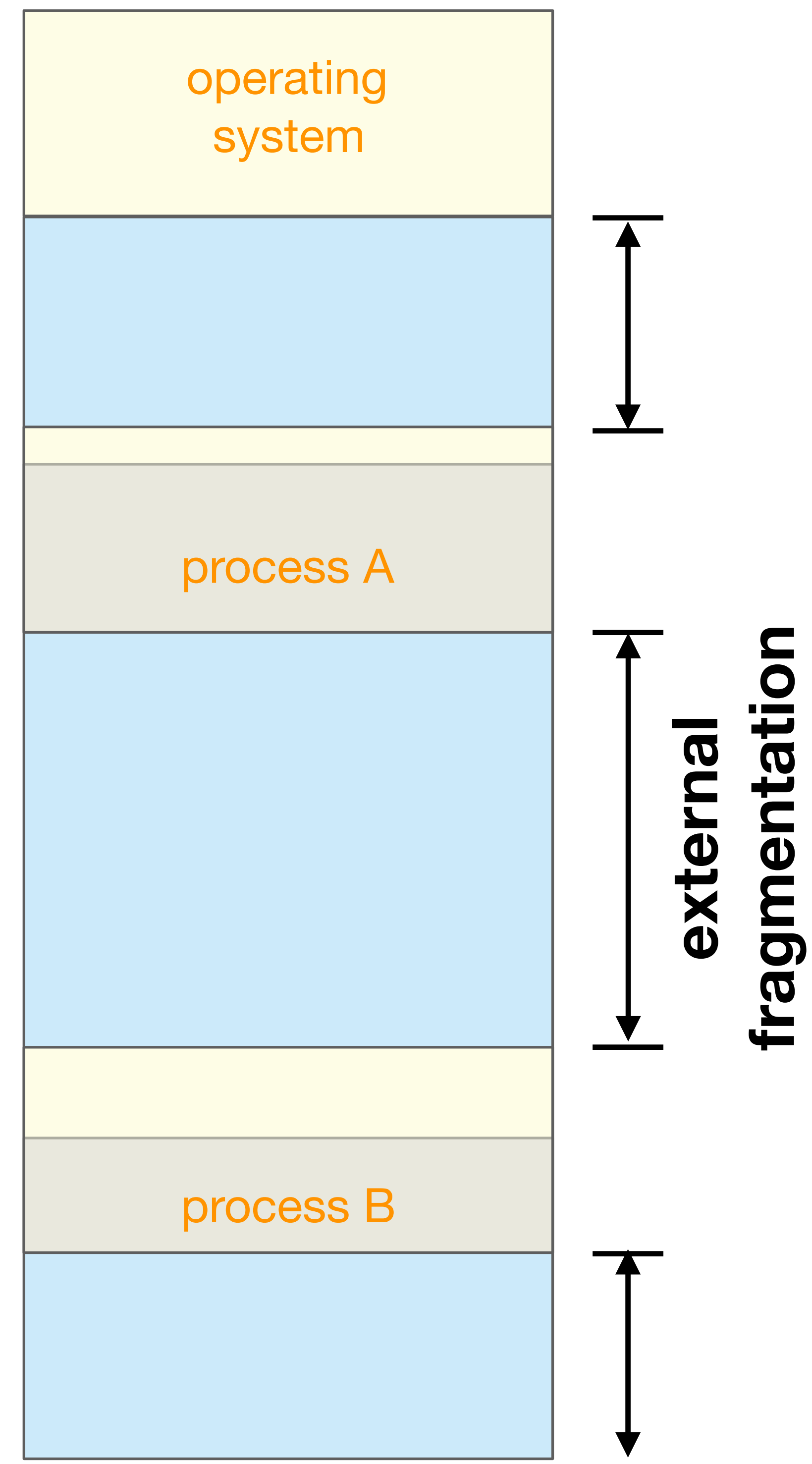
Memory allocation

- allocate a partition when a process is admitted into the system
  - which partition? depends on the allocation policy
- allocate a contiguous memory partition to the process
  - user variable-partition sizes for efficiency (sized to a given process' needs)
- OS keeps track of full blocks and empty blocks (i.e., holes)
  - using a linked list of empty blocks, for example



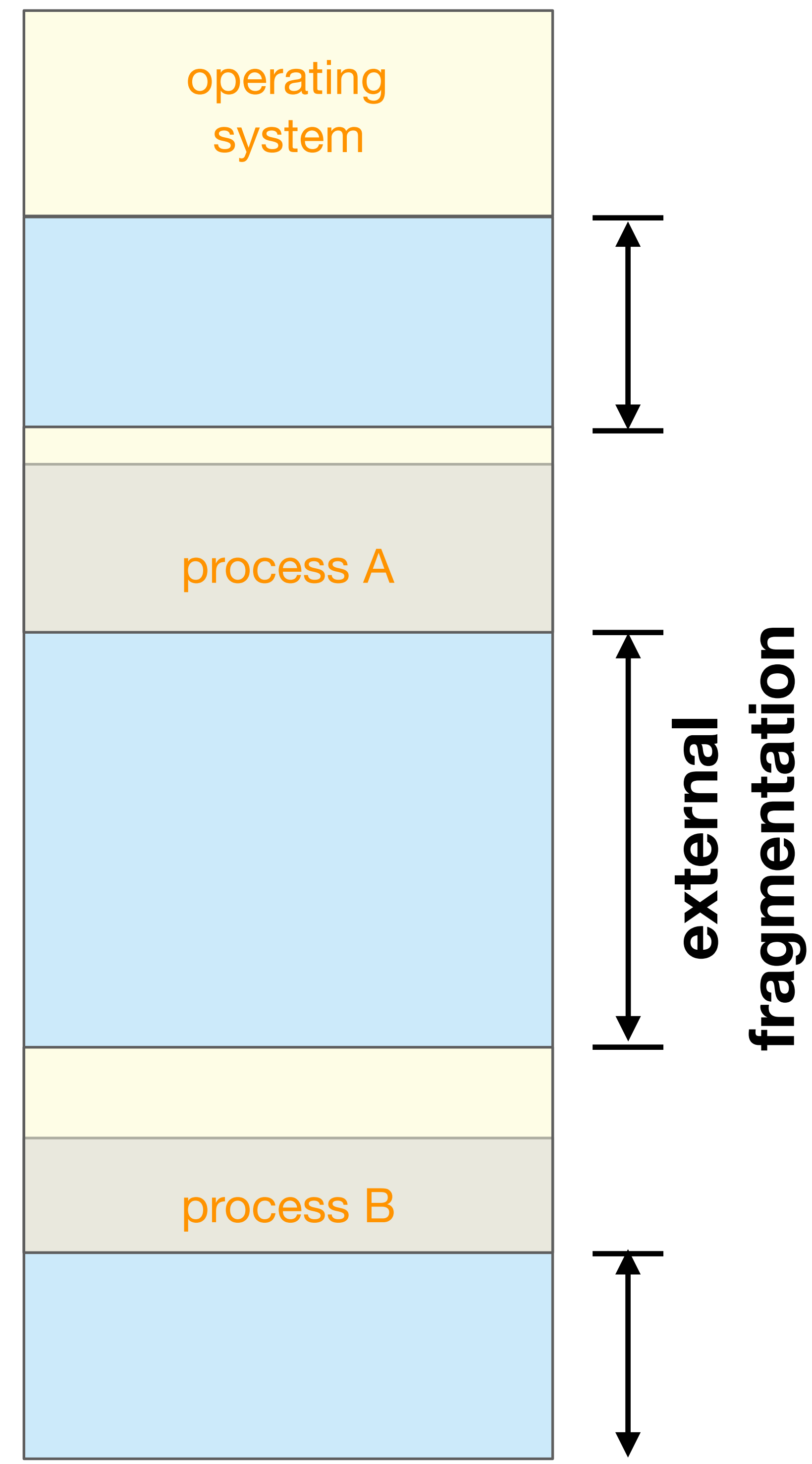
What is fragmentation?

- external fragmentation
  - unused memory between units of allocation
  - there is enough memory to fit a process in memory, but the space is not contiguous
  - **50-percent rule**: simulations show that N blocks are lost due to fragmentation for every 2N allocated blocks (33% wasted space)



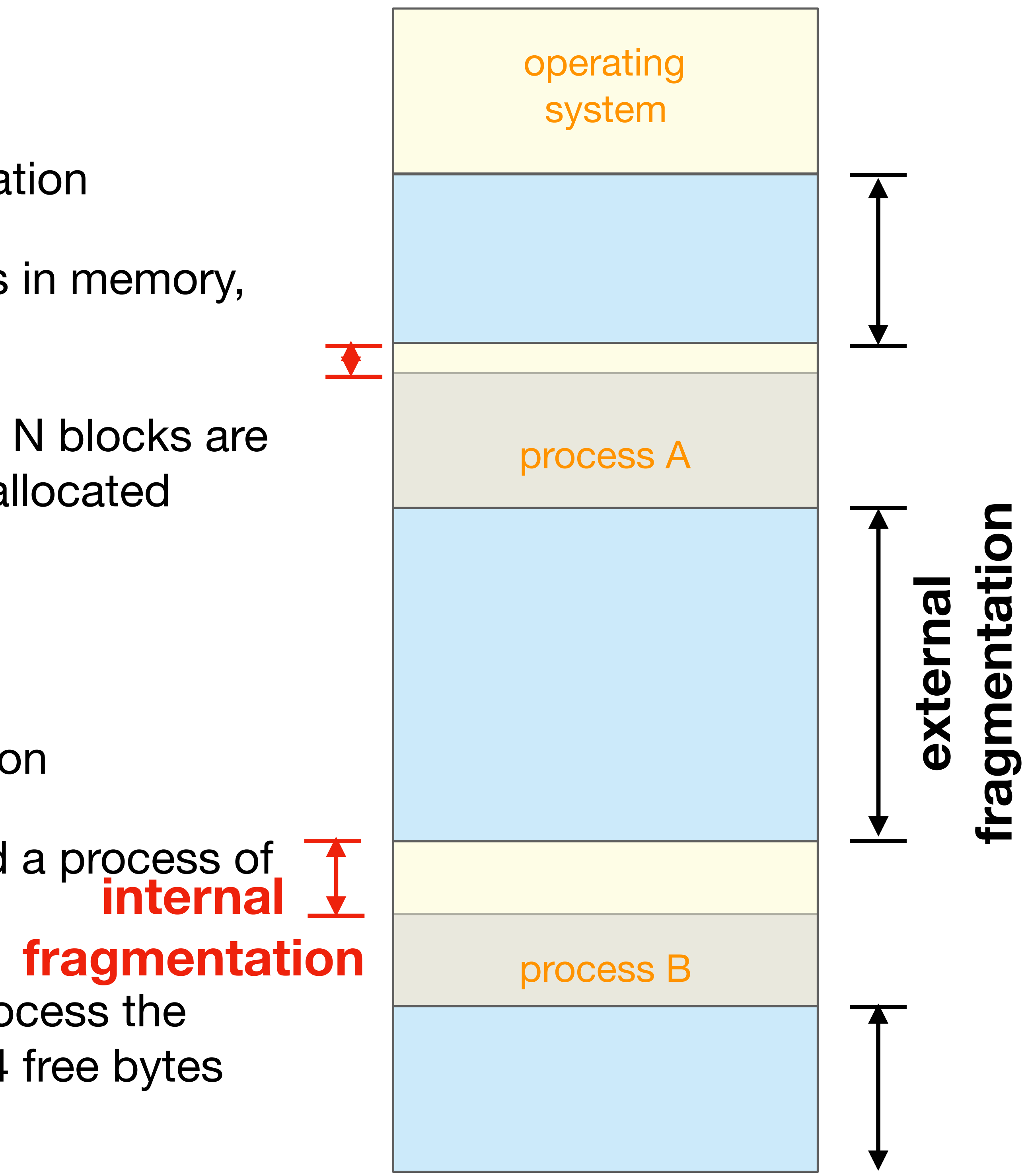
What is fragmentation?

- external fragmentation
  - unused memory between units of allocation
  - there is enough memory to fit a process in memory, but the space is not contiguous
  - **50-percent rule**: simulations show that N blocks are lost due to fragmentation for every 2N allocated blocks (33% wasted space)
- internal fragmentation
  - unused memory within a unit of allocation
  - consider a block size of 8192 bytes and a process of size 8128 bytes
    - it is more efficient to allocate the process the entire block than to keep track of 64 free bytes



What is fragmentation?

- external fragmentation
  - unused memory between units of allocation
  - there is enough memory to fit a process in memory, but the space is not contiguous
  - **50-percent rule**: simulations show that N blocks are lost due to fragmentation for every 2N allocated blocks (33% wasted space)
- internal fragmentation
  - unused memory within a unit of allocation
  - consider a block size of 8192 bytes and a process of size 8128 bytes
    - it is more efficient to allocate the process the entire block than to keep track of 64 free bytes



First-fit allocation

- use the first available free block larger than `n`
- implementation
  - requires a free block list sorted by address
  - allocation requires a search for the first partition that is large enough
  - deallocation requires checking if the freed blocks can be merged with other adjacent free blocks

First-fit allocation

- use the first available free block larger than `n`
- implementation
  - requires a free block list sorted by address
  - allocation requires a search for the first partition that is large enough
  - deallocation requires checking if the freed blocks can be merged with other adjacent free blocks
- advantages
  - simple implementation
  - results in larger free blocks towards the end of the address space
- disadvantages
  - slow allocation
  - external fragmentation

Best-fit allocation

- use the smallest available free block larger than `n` to minimize the size of external fragments produced
- implementation
  - requires a free block list sorted by size
  - allocation requires a search for the first partition that is large enough
  - deallocation requires checking if the freed blocks can be merged with other adjacent free blocks (more expensive than first fit)

Best-fit allocation

- use the smallest available free block larger than `n` to minimize the size of external fragments produced
- implementation
  - requires a free block list sorted by size
  - allocation requires a search for the first partition that is large enough
  - deallocation requires checking if the freed blocks can be merged with other adjacent free blocks (more expensive than first fit)
- advantages
  - works best if most allocations are small sized
- disadvantages
  - slow deallocation
  - external fragmentation
  - tends to produce many tiny fragments

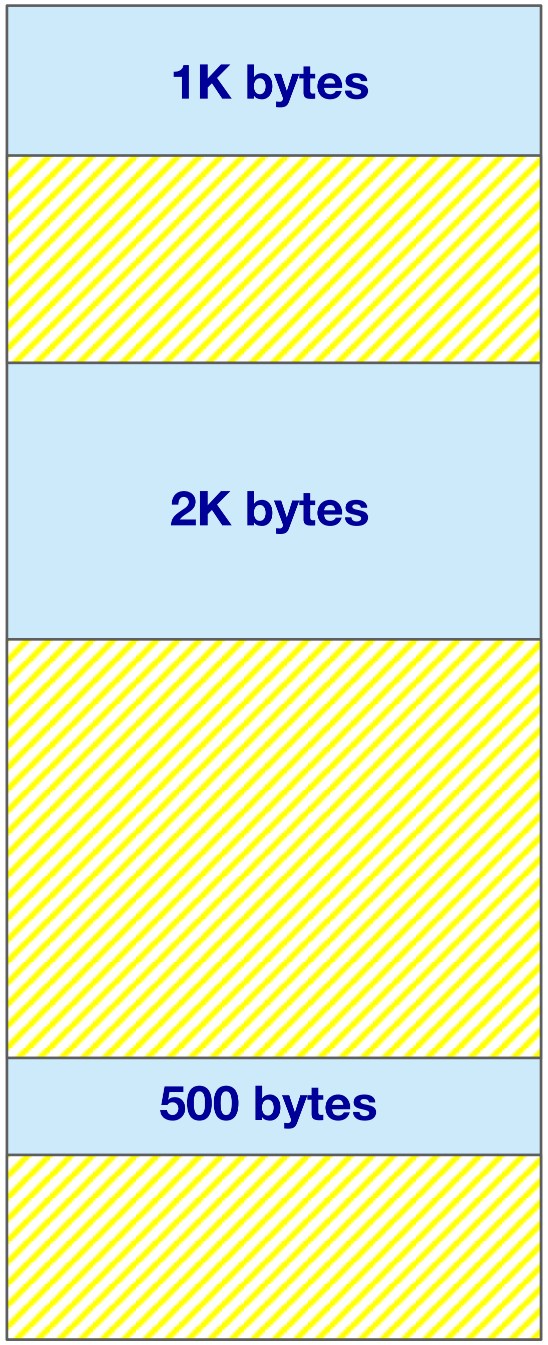
Worst-fit allocation

- use the largest available free block larger than `n` to avoid having too many tiny fragments
- implementation
    - requires a free block list sorted by size (descending)
    - allocation requires a search for the first (largest) partition
    - deallocation requires checking if the freed blocks can be merged with other adjacent free blocks (more expensive than first fit)

Worst-fit allocation

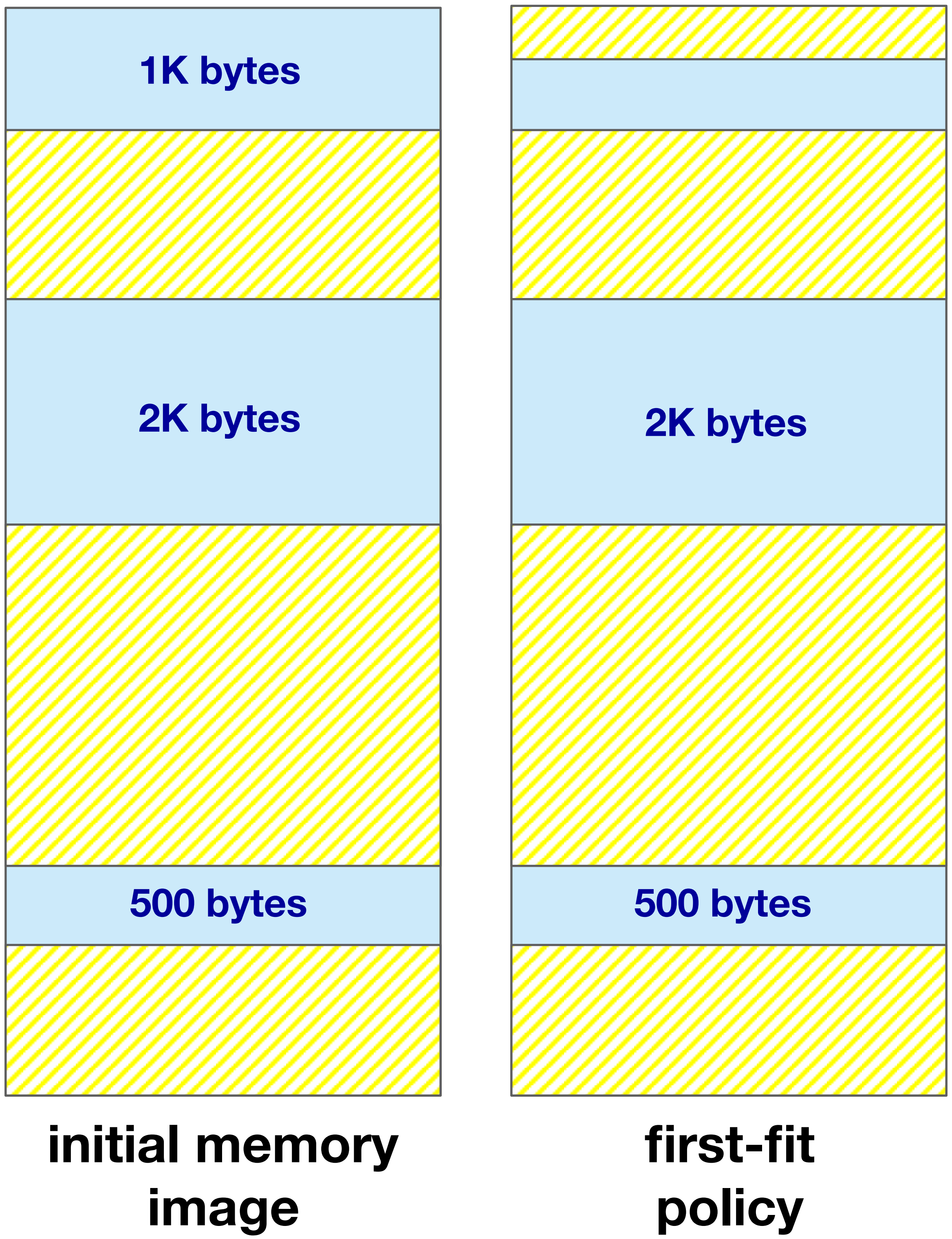
- use the largest available free block larger than `n` to avoid having too many tiny fragments
- implementation
  - requires a free block list sorted by size (descending)
  - allocation requires a search for the first (largest) partition
  - deallocation requires checking if the freed blocks can be merged with other adjacent free blocks (more expensive than first fit)
- advantages
  - works best if most allocations are medium sized
- disadvantages
  - slow deallocation
  - external fragmentation
  - may run into problem allocating large partitions because large free blocks are used for smaller allocations

How to allocate 300 bytes?

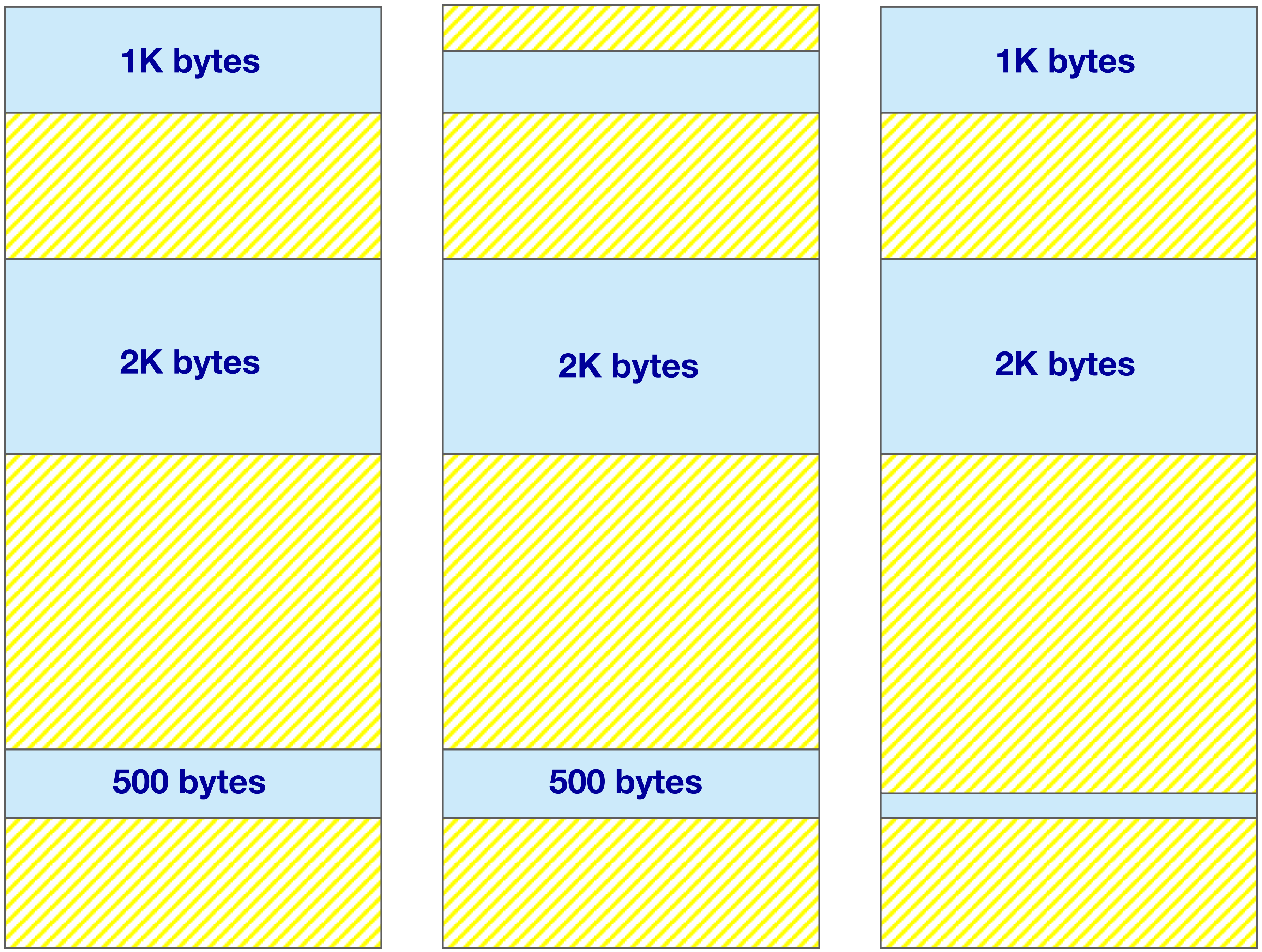


initial memory image

How to allocate 300 bytes?

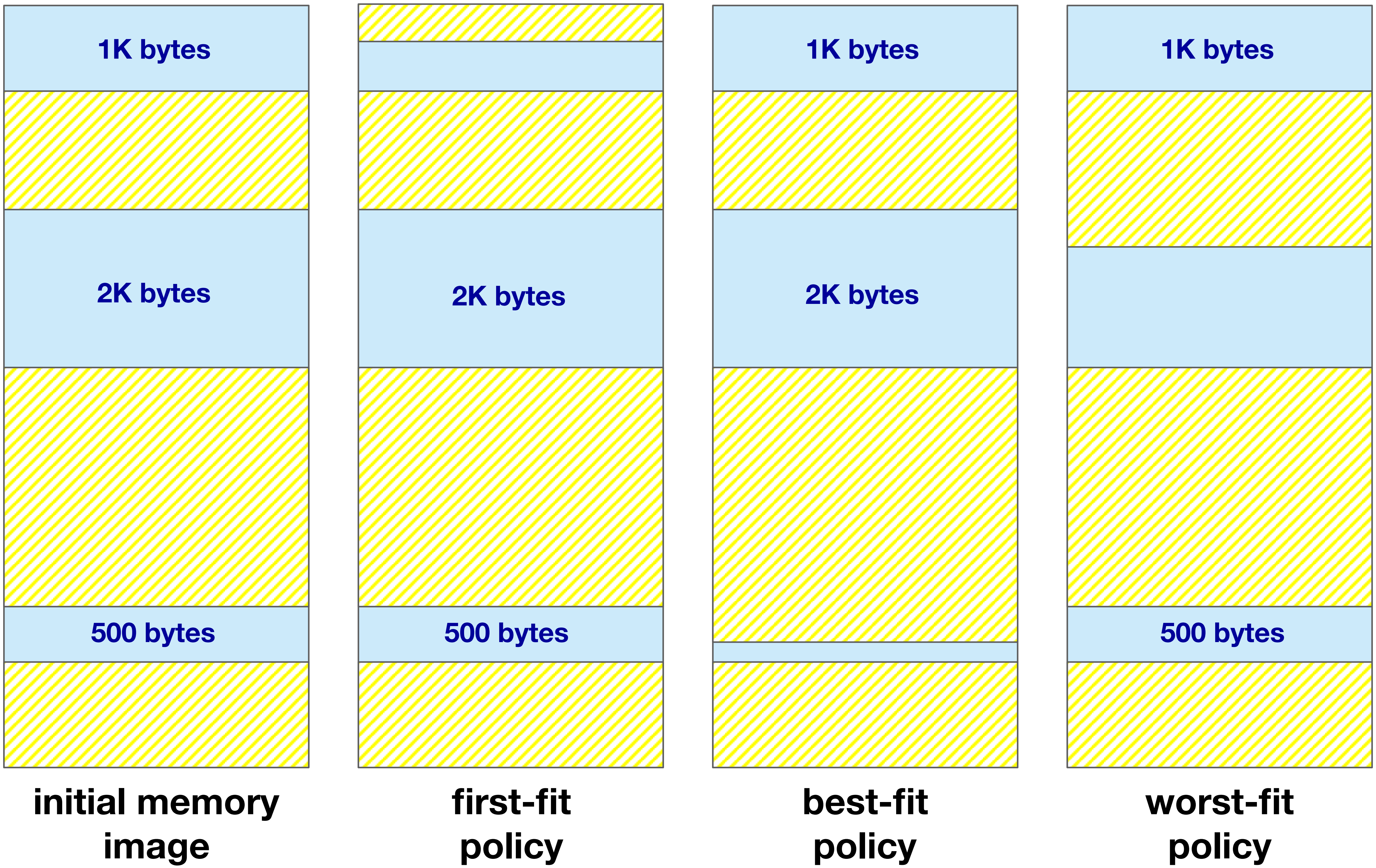


How to allocate 300 bytes?



initial memory image	first-fit policy	best-fit policy

How to allocate 300 bytes?

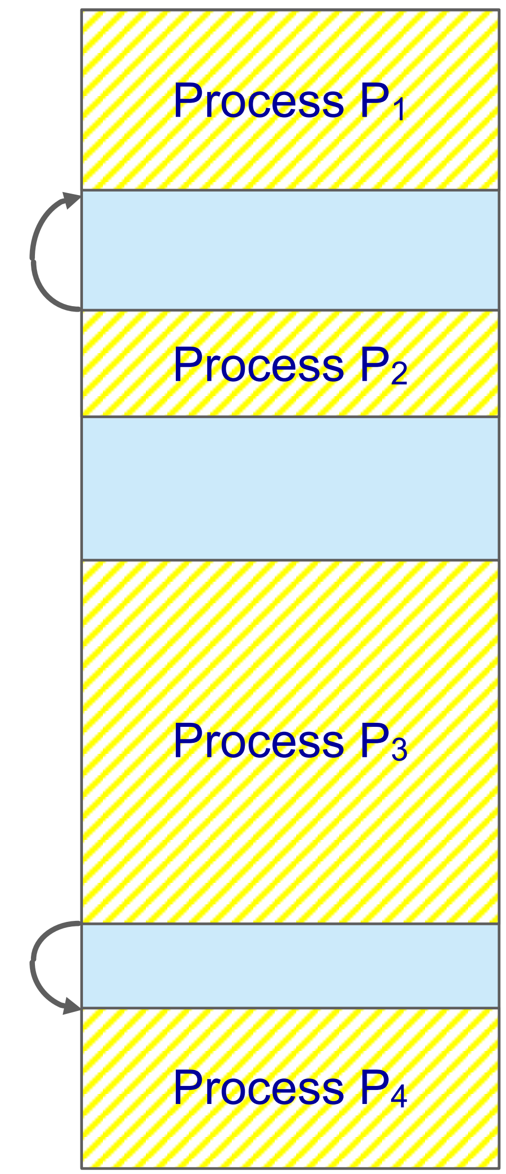


What's a good allocation policy?

- a policy that minimizes wasted space due to fragmentation
- simulations show first-fit and best-fit usually yield better storage utilization than worst-fit
  - first-fit is faster than best-fit

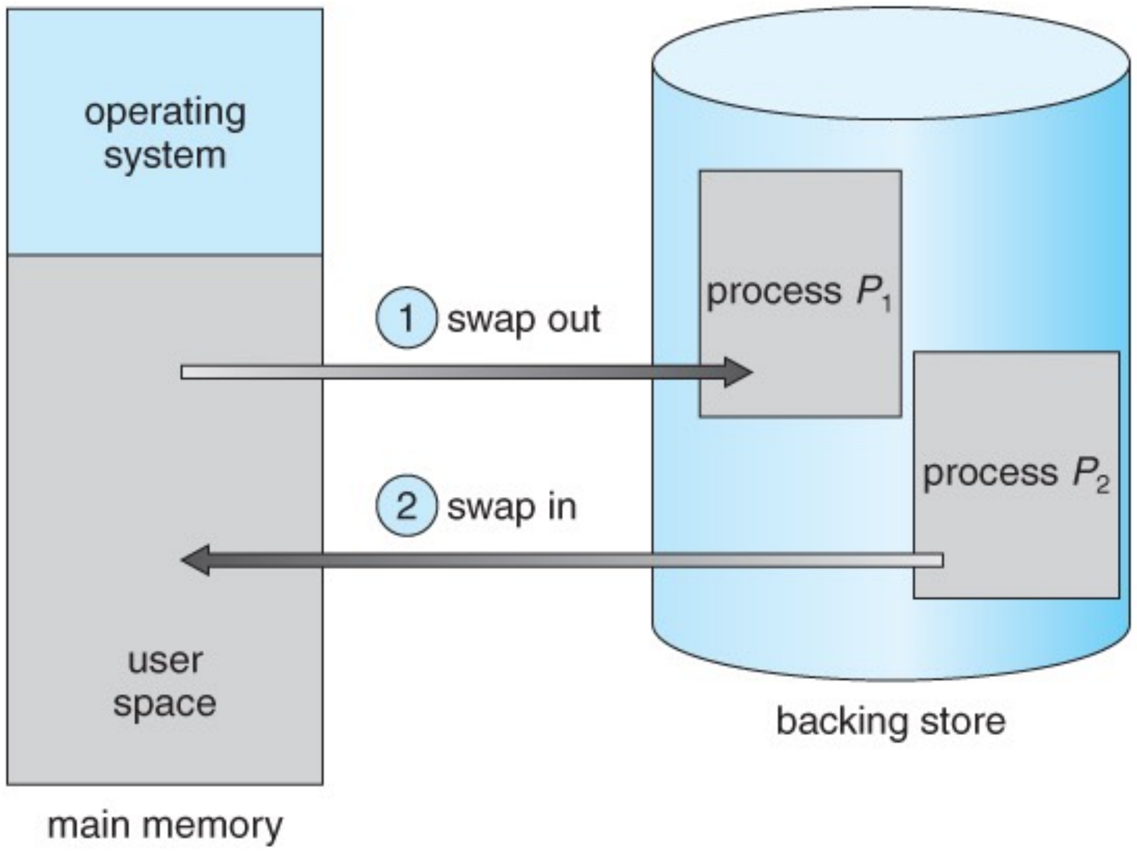
Compaction

- relocate processes to coalesce holes
  - the process shouldn't notice the change
  - requires logical addressed to be relocated at execution time
  - if addresses are relocated at load time, we cannot compact storage
- it can be done in different ways
  - how much memory is moved?
  - how big a hole is created?



What if not all processes fit into memory?

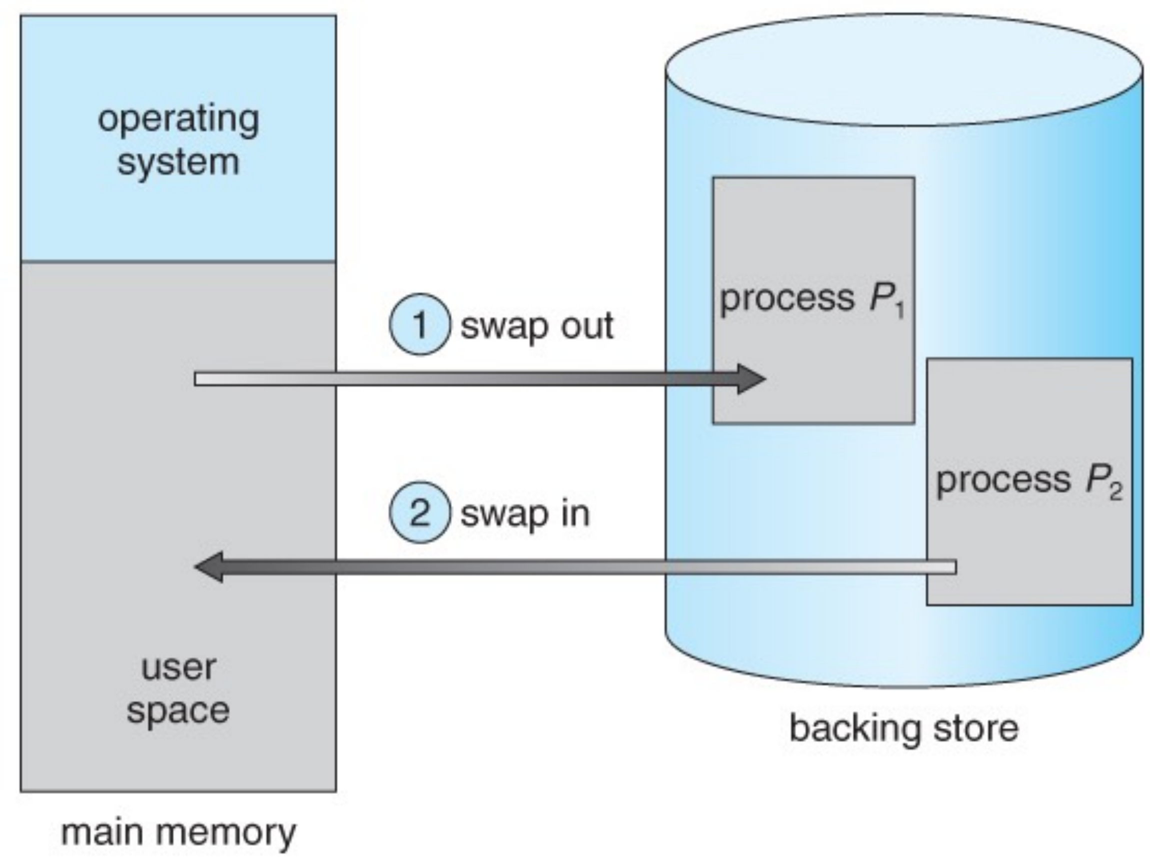
- swapping: preempt processes and reclaim their memory
  - frequency is dictated by the CPU scheduling algorithm



processes are copied from main memory to a backing store and are later copied back to main memory

What if not all processes fit into memory?

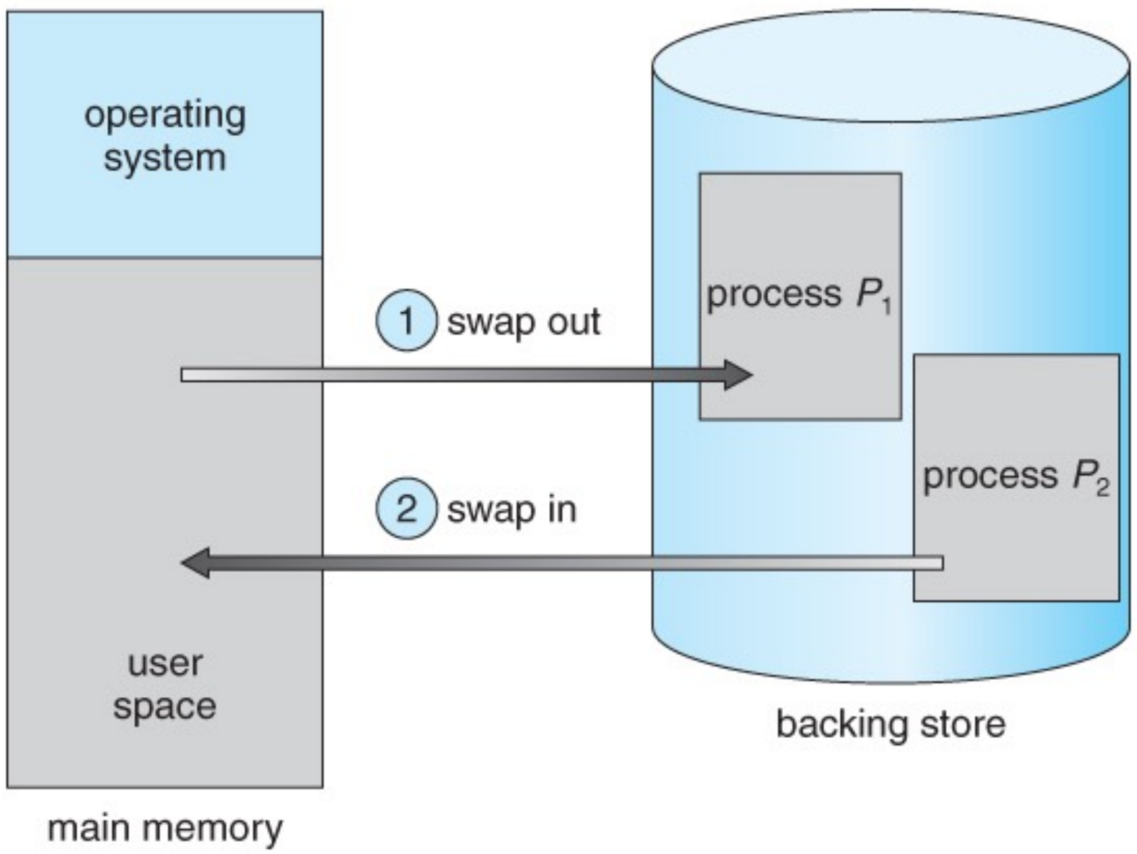
- swapping: preempt processes and reclaim their memory
  - frequency is dictated by the CPU scheduling algorithm
- when process becomes active again, the OS must reload it in memory
  - with static relocation, the process must be put in the same position
  - with dynamic relocation, the OS finds a new position in memory for the process and updates the relocation and limit registers



processes are copied from main memory to a backing store and are later copied back to main memory

What if not all processes fit into memory?

- swapping: preempt processes and reclaim their memory
  - frequency is dictated by the CPU scheduling algorithm
- when process becomes active again, the OS must reload it in memory
  - with static relocation, the process must be put in the same position
  - with dynamic relocation, the OS finds a new position in memory for the process and updates the relocation and limit registers
- compaction is easier with swapping



processes are copied from main memory to a backing store and are later copied back to main memory