Main Memory

ICS332

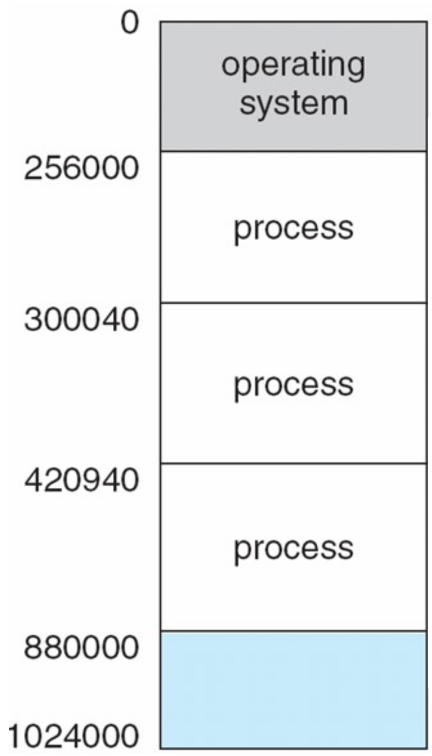
Operating Systems

Main Memory

- The OS must manage main memory because it manages processes that *share* main memory
- Main memory:
  - A large array of bytes (words), each with its own address
  - The memory unit sees a stream of addresses coming in on the memory bus
    - The memory unit is hardware
  - Each incoming address is stored in the memory-address register of the memory unit
    - Causing the memory unit to put the content at that address on the memory bus
- The CPU can only issue addresses that correspond to registers or main memory
  - There are no assembly instructions to directly read/write data to disk
- We're going to learn how the OS manages memory
  - Disclaimer: we'll describe how things work, then "break them", then describe how they really work, then "break them again" and so on until we get to how things really work (really)

Contiguous Memory Allocations

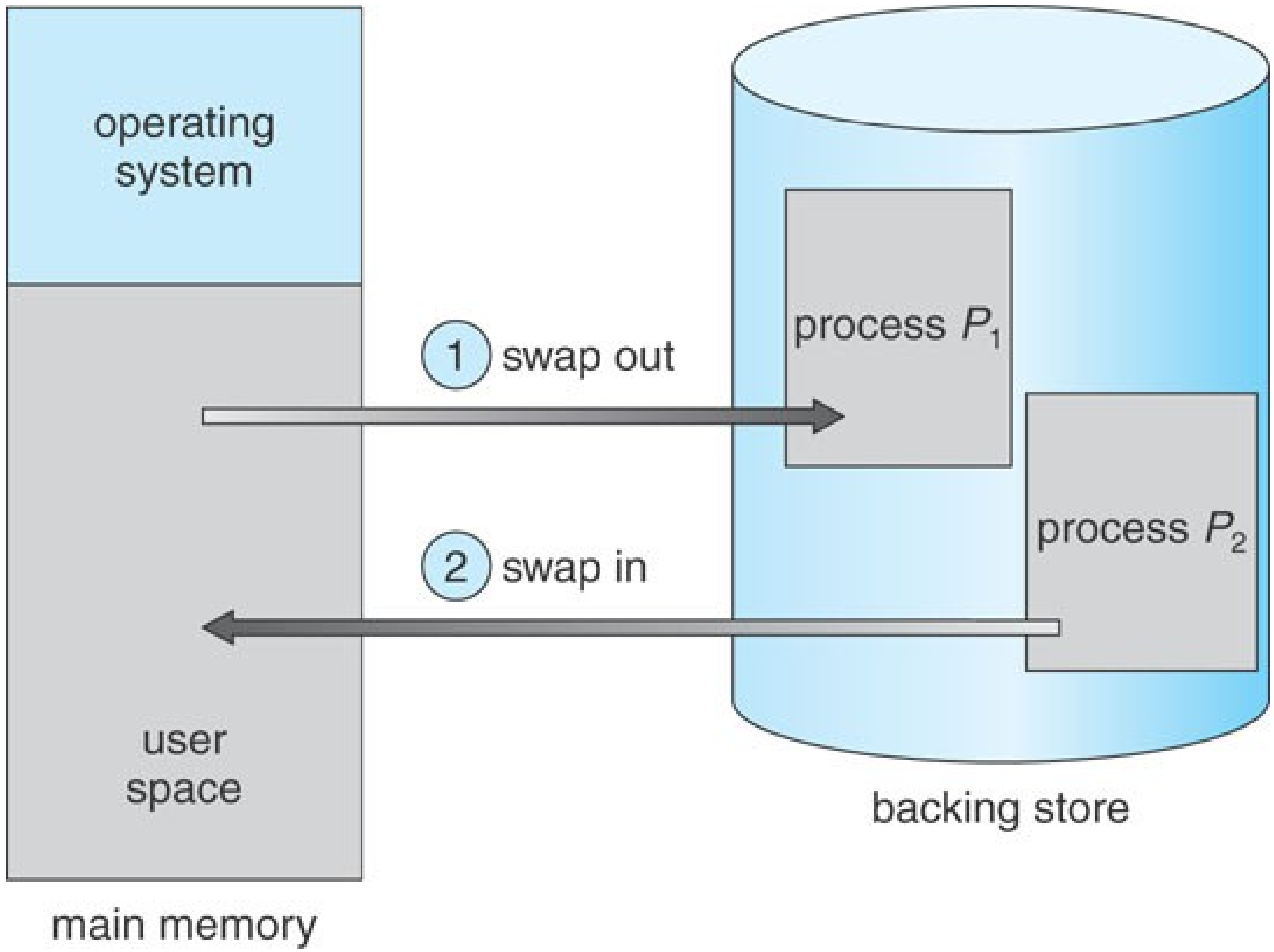
- Let’s assume what we’ve always assumed so far: each process is allocated a contiguous zone of physical memory



Swapping

- **Sometimes, not all processes can fit in memory**
- Some must be saved to a “backing store”, i.e., the disk
- Moving processes back and forth between main memory and the disk is called swapping
- When a process is swapped back in, it may be put into the same physical memory space or not
  - Enabled by address binding at execution time
- As we know, the OS maintains a ready queue of processes
- Some of the ready processes reside in memory, some on disk
- Whenever the OS says: “I give the CPU to process X”, then it calls the dispatcher, who loads X from disk if needed
  - And then does the usual register loads, etc.
- Consequence: some context switches can involve the disk

Swapping



operating system
1 swap out
process P_1
process P_2
2 swap in
user space
main memory
backing store

Swapping is slow

- Context-switching to a process in memory is fast
- Context-switching to a process on disk is really slow
  - Consider a process with 500MB address space
  - Consider a disk with 10ms latency and 50MB/sec bandwidth
  - The time to load the process is 10010ms
    - And the time to store the process is likely higher
- How do we cope with slow swapping?
  - We ask programs to tell us exactly how much memory they need (malloc and free are not there just to make your life difficult)
  - The OS may then opt to swap in and out smaller processes rather than larger processes
  - We may use a swap partition (as opposed to a file) so as to minimize expensive disk seeks (more on this much later)

Swapping and I/O

- Swapping a process to disk may require that the process be completely idle, especially for I/O purposes
- If a process is engaged in I/O operations, these operations could be asynchronously writing data into the process' address space
  - e.g., DMA
- If we swap it out and replace it by another process, that other process may see some of its memory overwritten by delayed I/O operations!
- Two solutions:
  - Never swap any process that has pending I/O
  - Do all I/O in kernel buffers, and then the kernel can decide what to do with it

Swapping in OSes

- Swapping is often disabled
- Swapping is done only in “exceptional” circumstances
  - e.g., a process has been idle for a long time and memory space could be used for another process
  - e.g., the load on the system is getting too high
- In Windows 3.1, swapping was user-directed!
  - The user decides to allow/deny swapping in and out
- If the normal mode of operation of the system requires frequent swapping, you’re in trouble
- Modern OSes do not always swap whole processes
  - Say you have 3GB of available RAM and two 1.6GB processes
  - It would seem to make sense to keep one process and 94% of the other in RAM at all time
  - This is called “paging” (see later in these slides)

Smaller Address Spaces

- Swapping is slow, and the more it is avoided, the better
- So there is a strong motivation to make address spaces as small as possible
  - Intuitively, if one can save RAM space then one should do it
- Two techniques are used to reduce the size of the address spaces
  - Dynamic Loading
  - Dynamic Linking

Dynamic Loading

- One reason for an address space being large is that the text segment is itself very large
  - i.e., many lines of code
- But often large amounts of code are used very rarely
  - e.g., code to deal with errors, code to deal with rarely used features
- With dynamic loading, the code for a routine is loaded only when that routine is called
  - All dynamically loadable routines are kept on disk using a relocatable format (i.e., the code can be put anywhere in RAM when loaded)

Dynamic Loading

- Dynamic loading is the responsibility of the user program
  - The OS is not involved, although it can provide convenient tools to ease dynamic loading
- Example: Dynamic Loading in Java
  - Java allows classes to be loaded dynamically
  - The ClassLoader class is used to load other classes
  - Simple example, for a loaded class named “MyLoadedClass”, which has a “print” method that takes as input a String
    - ClassLoader myClassLoader = ClassLoader.getSystemClassLoader()
    - Class myLoadedClass = myClassLoader.loadClass(“MyLoadedClass”)
    - Object instance = myLoadedClass.newInstance()
    - Method method = myLoadedClass.getMethod(“print”, new Class[] {String.class})
    - method.invoke(instance, new Object[] {“input string”})

Dynamic Linking

- The default: static linking
  - All libraries and objects are combined into one (huge) binary program
- Dynamic Linking is similar in concept to dynamic loading, but here it's the linking that's postponed until runtime
- We call such libraries: shared libraries
- When dynamic linking is enabled, the linker just puts a *stub* in the binary for each shared-library routine reference
- The stub is code that
  - checks whether the routine is loaded in memory
  - if not, then loads it in memory
  - then replaces itself with a simple call to the routine
    - future calls will be "for free"

Dynamic Linking

- So far, this looks a lot like Dynamic Loading
  - In fact, better, because more automated
- BUT, all running processes can share the code for the dynamic library thus saving memory space
  - which is why it's called a shared library (.so, .dll)
- So, for instance, the code for "printf" is only in one place in RAM
- This is also very convenient to update a library without having to relink all programs
  - Just replace the shared library file on disk, an new processes will happily load the new one
    - Provided the API hasn't changed of course
- Dynamic Linking requires help from the OS
  - To break memory isolation and allow shared text segments
  - This comes "for free" with virtual memory as we'll see

Looking at Shared Libraries

- On Linux system the ldd command will print the shared libraries required by a program
  - turns out, no need to use strace after all (Prog. Assignment #1)
- For instance, let's look at the shared libraries used by /bin/ls, /bin/date
  - The compiler adds stuff in the executable so that ldd can find this information and display it
- When you run this program, all those libraries are loaded into memory if not already there
- Turns out, on Linux, you can override functions from loaded shared libraries by creating yourself a small shared library
- Let's try this...
  - Inspired by the "Overriding System Functions for Fun and Profit" post at hackerboss.com (by "Ville Laurikari")

Overriding calls

- Let’s modify what /bin/date does
- As seen in the ldd output, /bin/date uses libc.so.6, the standard C library
  - In fact every program uses this!
  - It had better not be replicated in memory for each process!
- By looking at the code of the C library (which is open source), you can figure out how to write your own version of a few functions as follows
- Let’s look at our “replacement” code, to print the time one hour ago
  - Based on overriding the *localtime* function in libc
  - man localtime (converts a number of seconds since some time in the past to a data structure that describes localtime)

Slow by 1 hour

Access to tons of GNU/Linux things that are not part of the C standard (in our case, dynamic loader functionality)

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Slow by 1 hour

_init() in a shared library is executed when the library is loaded

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep [illegible] 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Slow by 1 hour

Our _init() function first prints a message

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Slow by 1 hour

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time

struct tm *localtime(const time_t *time
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Our _init() function then finds the address of the *localtime* function in one of the dynamic libraries loaded after this one, i.e, in libc.so.6.
Once the address is found, then is is stored in function pointer orig_localtime.
We do this so that we can call the original *localtime* function in our replacement *localtime* function

Slow by 1 hour

Our replacement for the original localtime function found in libc.so.6
This function takes a pointer to an integer-like number of seconds elapsed since Jan 1st 1970

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Slow by 1 hour

Compute the date 3600 seconds ago by modifying the number of second elapsed since the beginning of the epoch

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Slow by 1 hour

Call the original *localtime* function which is now “faked” into thinking that the number of seconds elapsed since the beginning of the epoch is 3600 seconds less than it really is

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading a weird date.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Let's try it...

- Compiling it
  - gcc -fPIC -DPIC -c weirddate.c
- Creating the shared library
  - ld -shared -o weirddate.so weirddate.o -ldl
- Running it
  - date
  - export LD_PRELOAD=./weirddate.so
    - To create an environment variable
  - date

Overriding with LD_PRELOAD

- This turns out to be VERY useful
- Let’s say you want to write a way to count heap space usage for any executable
  - Write modified malloc() and free() versions that call the original malloc() and free() functions but that record a total count of allocated bytes and print it
- Tons of other “serious” usages when you want to add something to an existing library function, or do something totally different

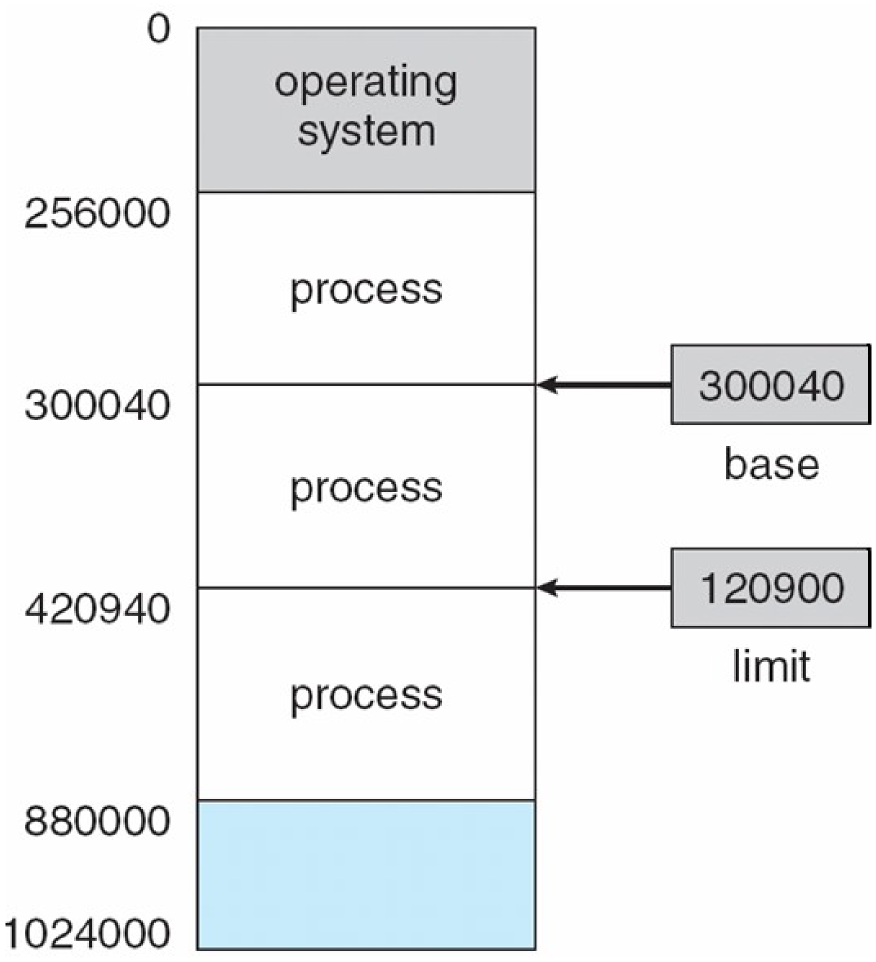
Onward to RAM management

- For now, let’s consider that each process consists of a “slab” of memory, without thinking about dynamic loading/linking
- A process should only access data in its own address space
- How does the OS/hardware enforce this memory protection?

Simple Memory Protection

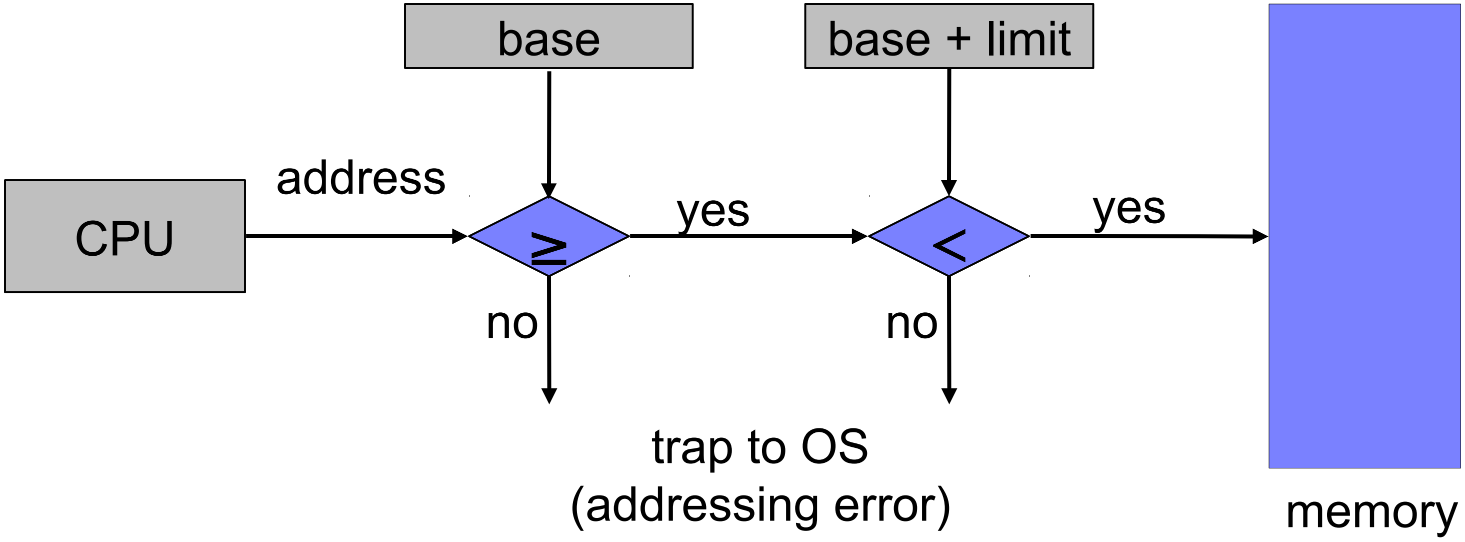
- To provide memory protection, one must enforce that the range of addresses issued by a process is limited
- This is done by the OS with help from the hardware
- The simplest approach: use two registers
  - Base Register: first address in the address space
  - Limit Register: length of the address space

Base and Limit Registers



Base and Limit Registers

- When "giving" the CPU to a process, the OS sets the values for the two registers
  - Done by the dispatcher component, during the context switch
- Setting these values is done via privileged instructions
  - For obvious reasons
- Then, the **hardware** uses these values:



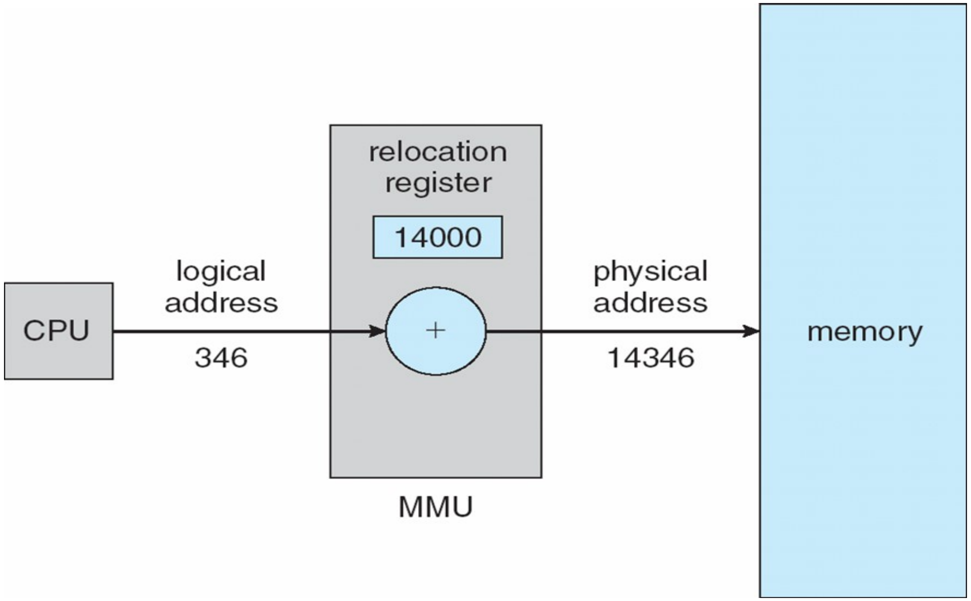
What's the problem?

- The setup on the previous slide works
- But it would be really convenient for the programmer/compiler to not have to care about where the program is in physical memory
- It would be great if I could think of my addresses as going from 0 to some maximum
  - The system should hide from me my actual location in physical memory
  - That way my program can be moved around in memory and that should all be handled by the OS not by the programmer
  - This is called relocatable code

Logical vs. Physical Addresses

- Let’s call an address that’s put in the memory unit’s memory-address register a physical address
- Let’s call an address generated by the CPU a logical address
- A program references a logical address space, which corresponds to a physical address space in the memory
  - logical address = virtual address
    - And we use both terms interchangeably
- Logical addresses are between 0 and some maximum
- There is a translation from logical to physical addresses
  - Done by the memory-management unit (MMU), a piece of hardware
- This is very simple to achieve...

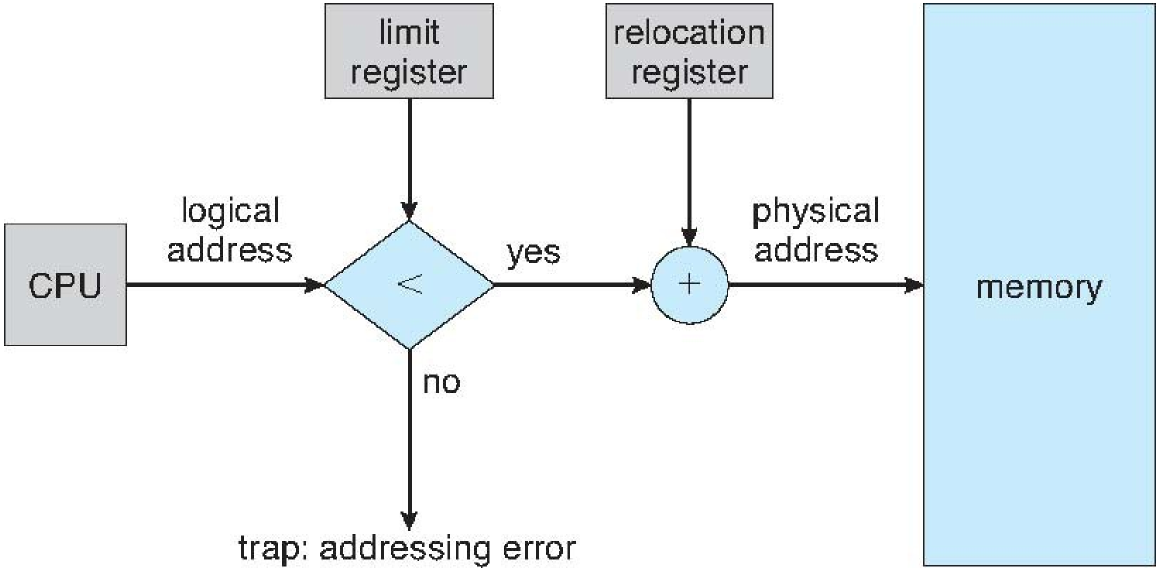
Simple Logical-to-Physical



- One option: a relocation register added to all logical address
  - Equivalent to the “base register” from a few slides ago
- The program works with logical addresses
  - In the range 0 to max
- The program never “sees” physical addresses
  - In the range R to (R+max)
- We just need to enforce that the logical addresses are between 0 and max...

Relocation and Limit Registers

- Relocation register: smallest valid physical @'s
- Limit register: range of valid logical @'s (the "max")



- Loaded by the dispatcher during a context-switch
- Used to provide both protection and relocation
  - Moving a process: memcopy it and update the relocation reg.

We now have the mechanism…

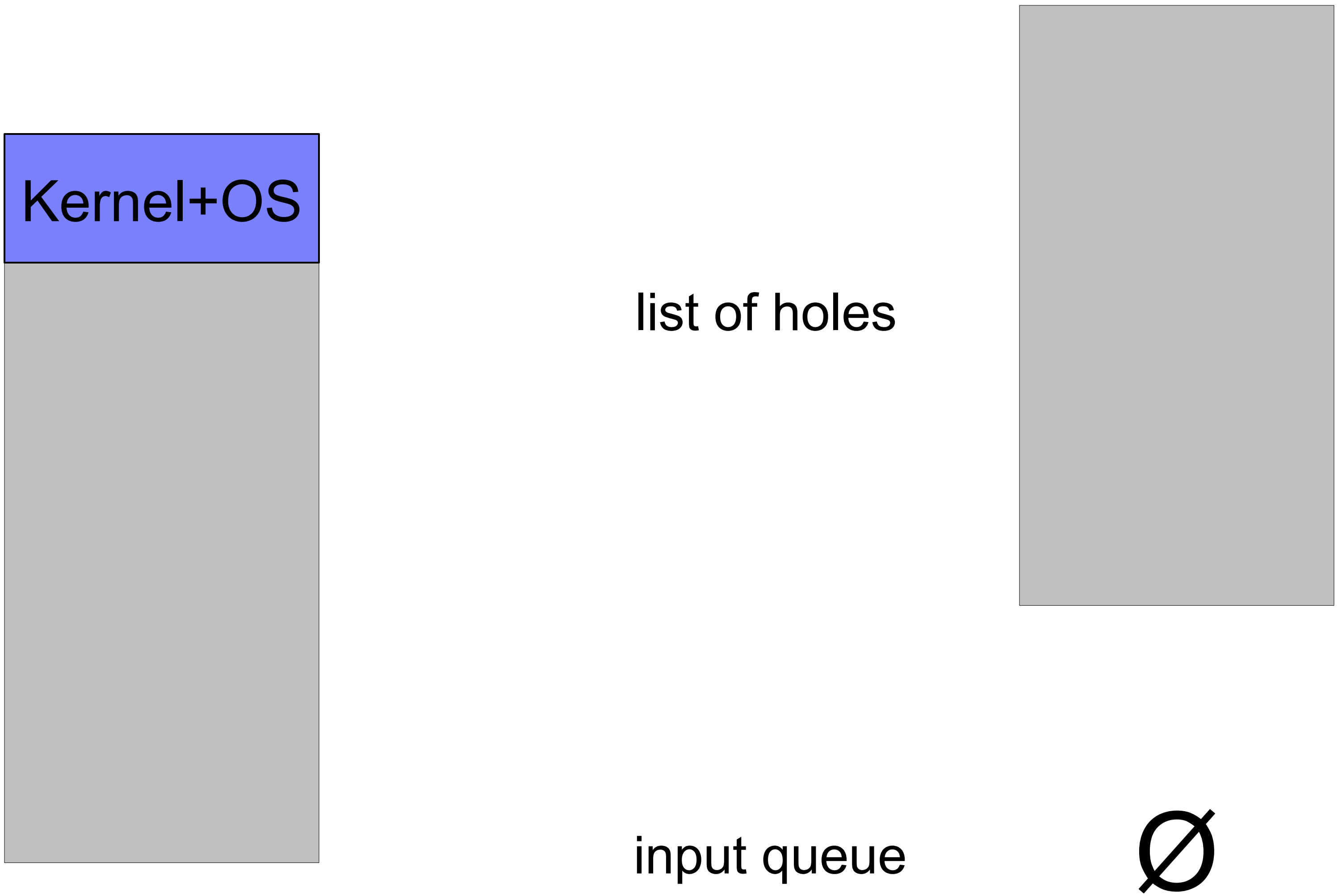
- We have the mechanism to allocate each process a “slab” of RAM, and to have it issue addresses that fall in that slab
  - Or rather, to detect when it issues addresses outside of the slab and terminate it

- Question: What’s the policy?
  - How do I decide where to place each slab in RAM?

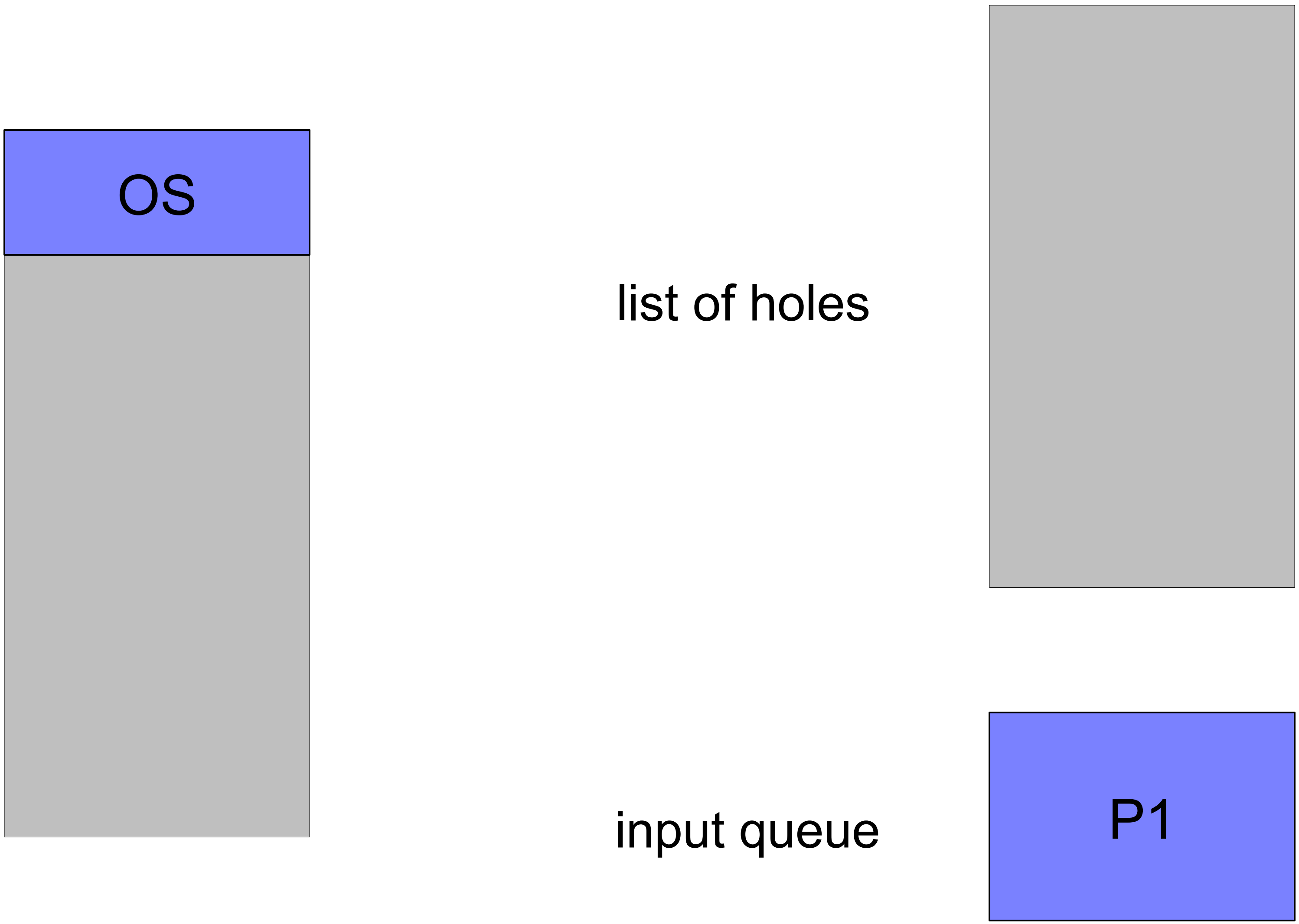
Memory Partitioning

- Where do we put processes in memory?
- The Kernel can keep a list of available memory regions, or holes
  - The list is updated after each process arrival and departure
- When a process enters the system, it’s put on an input queue
  - The “I am waiting for memory” queue
- Given the list of holes and the input queue, the Kernel must make decisions
  - Pick a process from the input queue
  - Pick in which hole the process is placed
- This is the dynamic storage allocation problem
- Goal: allow as many processes in the system as possible
- Unfortunately it is a theoretically difficult problem
  - And it’s an “on-line” problem (i.e., we don’t know the future)

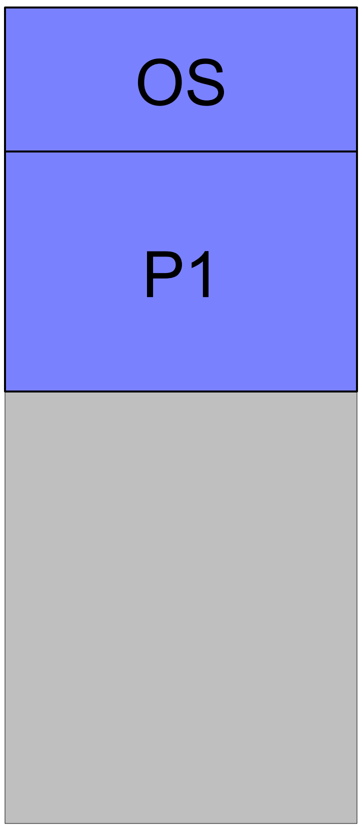
Memory Partitioning



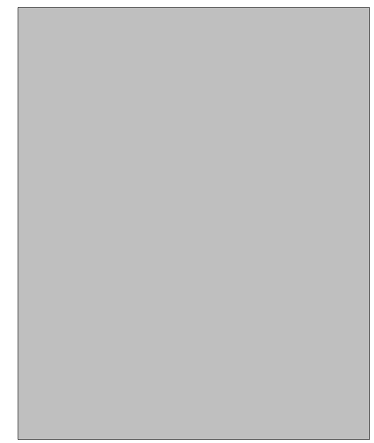
Memory Partitioning



Memory Partitioning



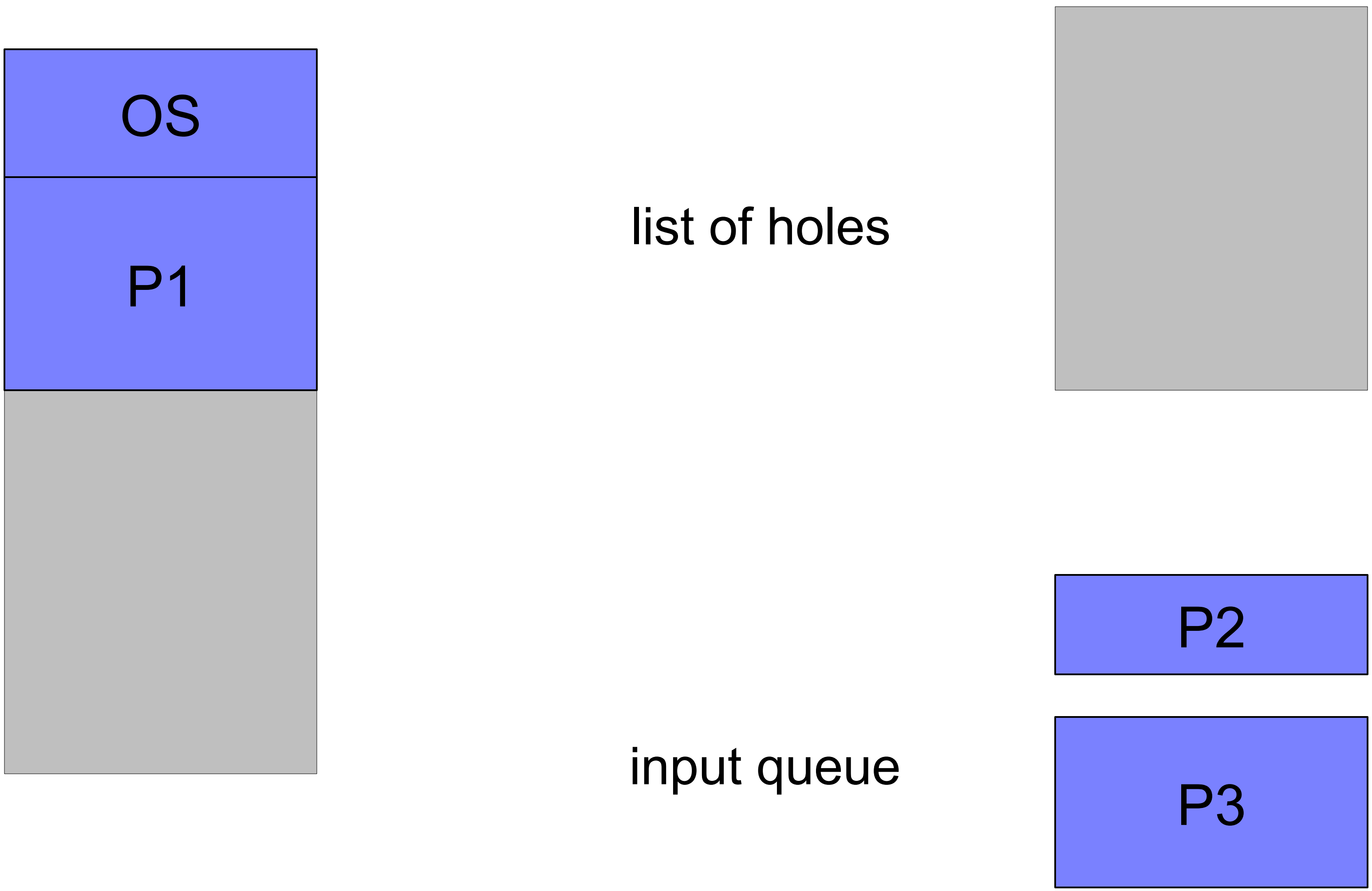
list of holes



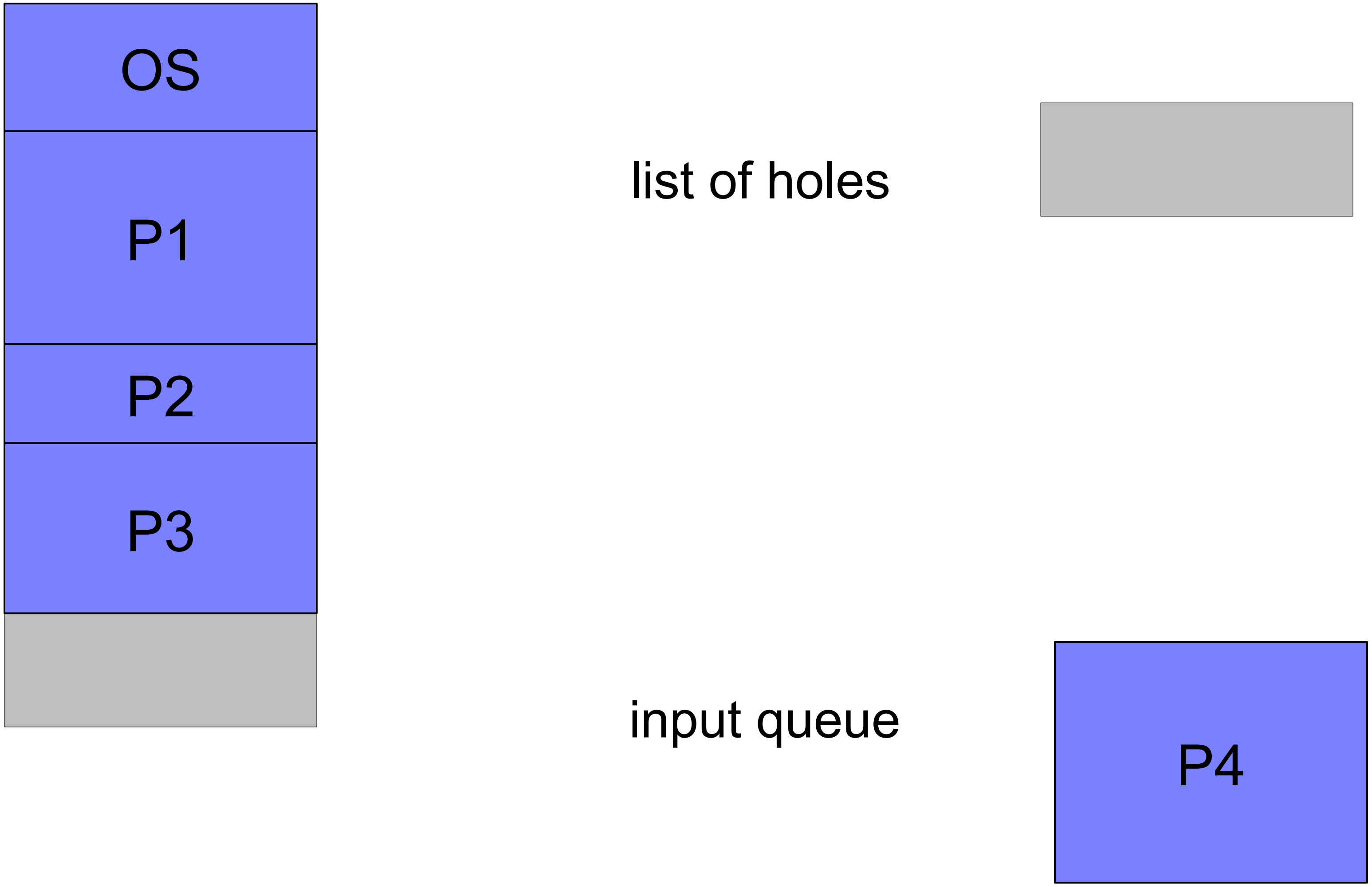
input queue

Ø

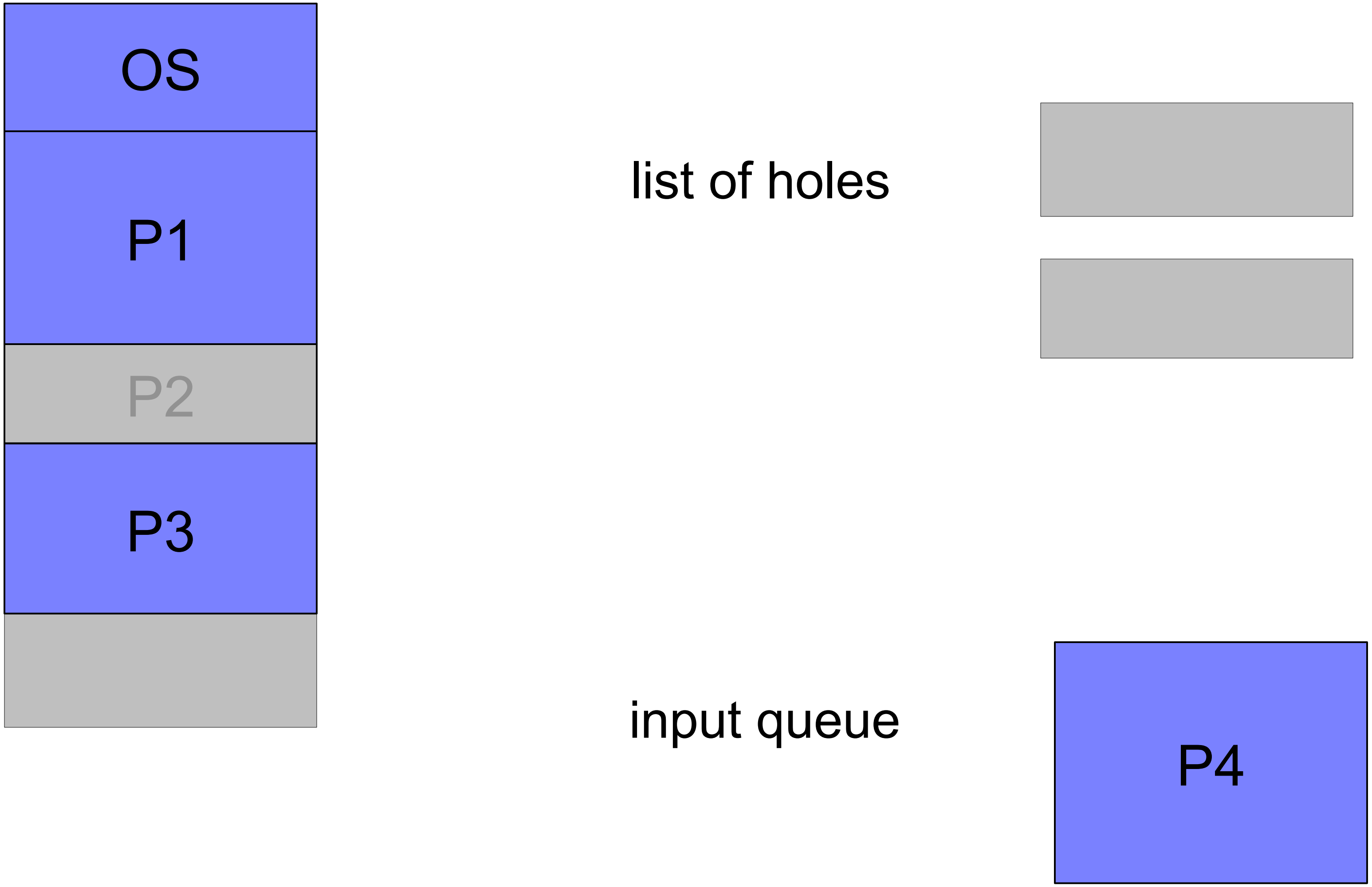
Memory Partitioning



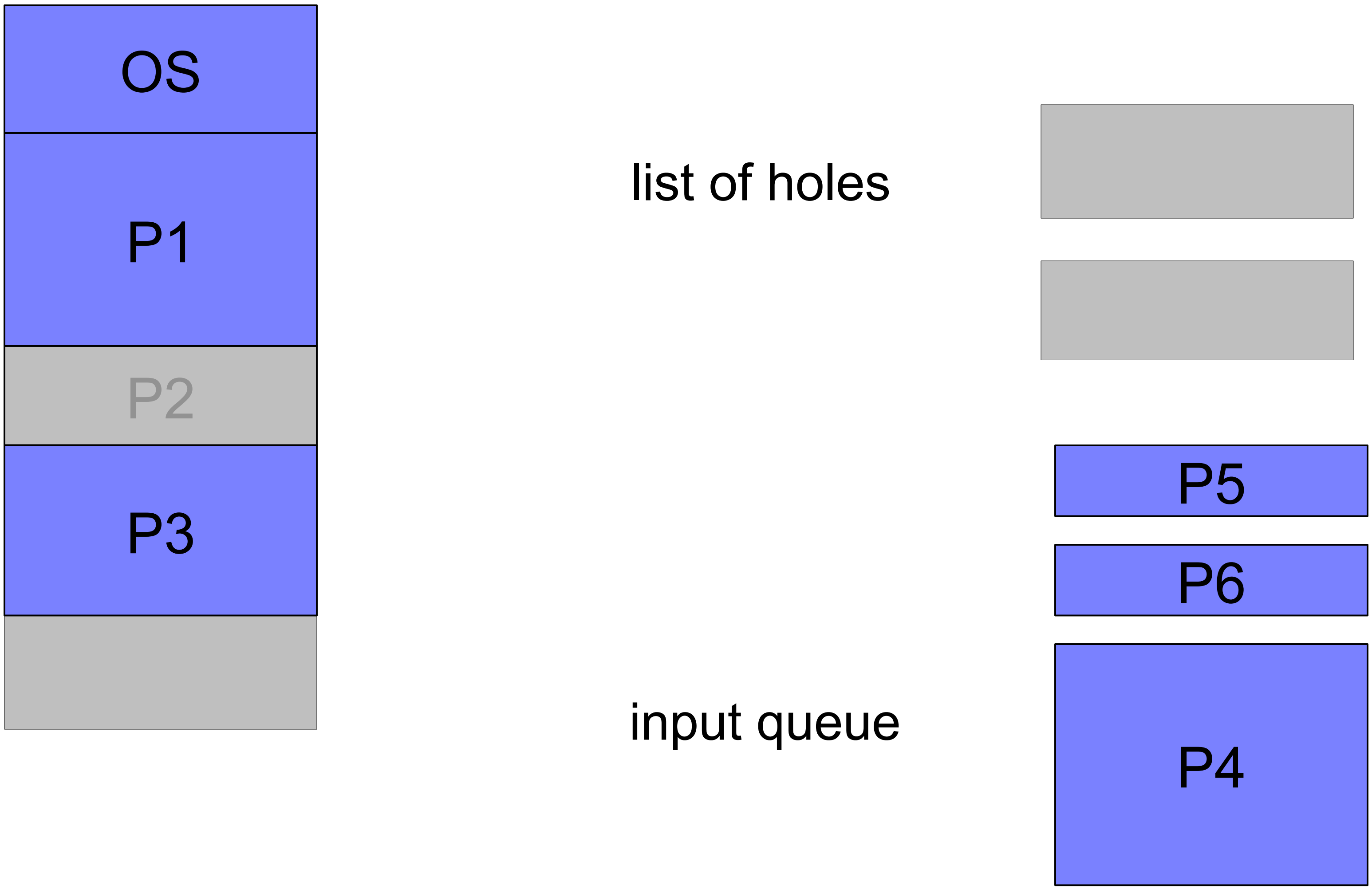
Memory Partitioning



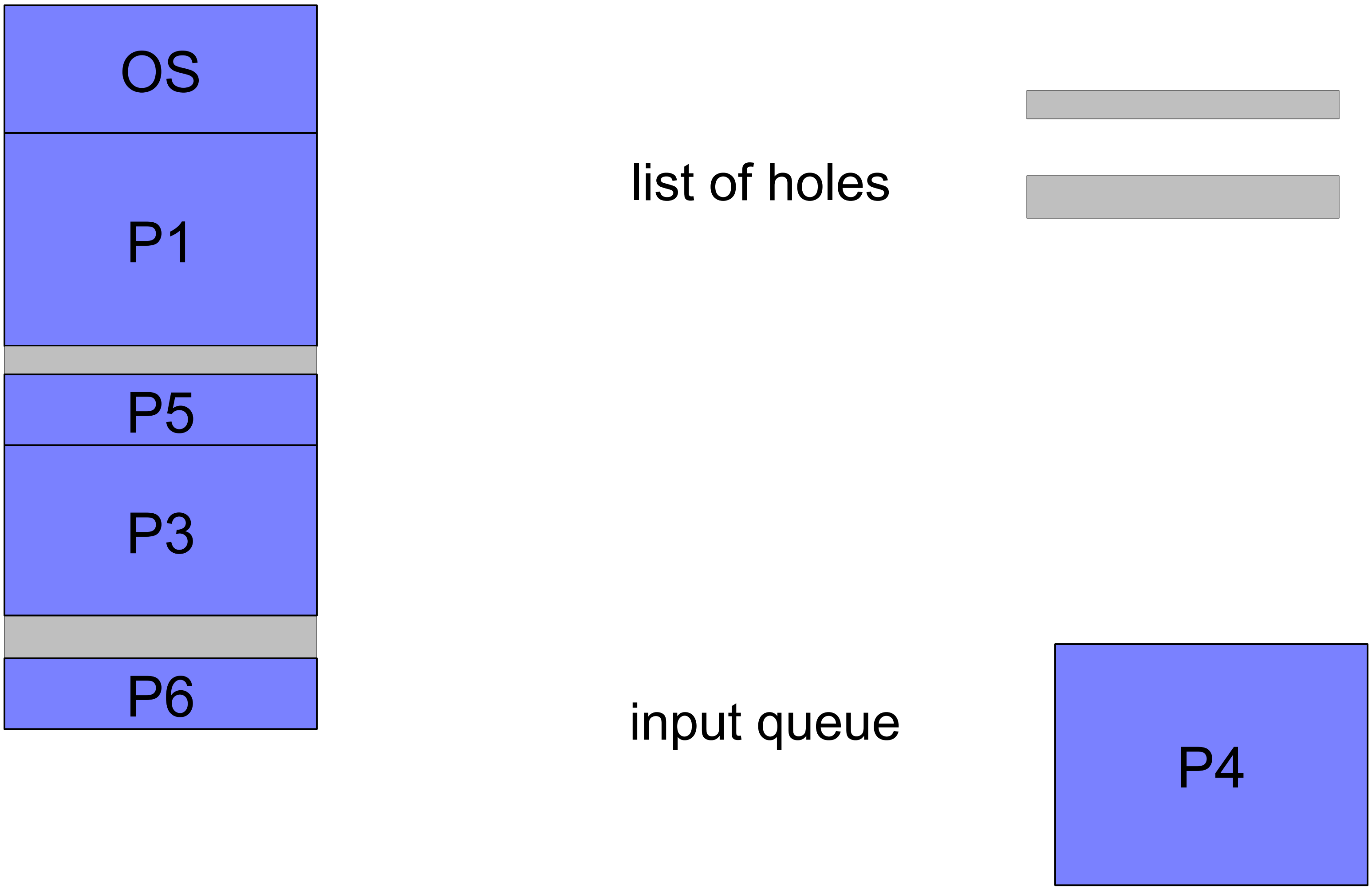
Memory Partitioning



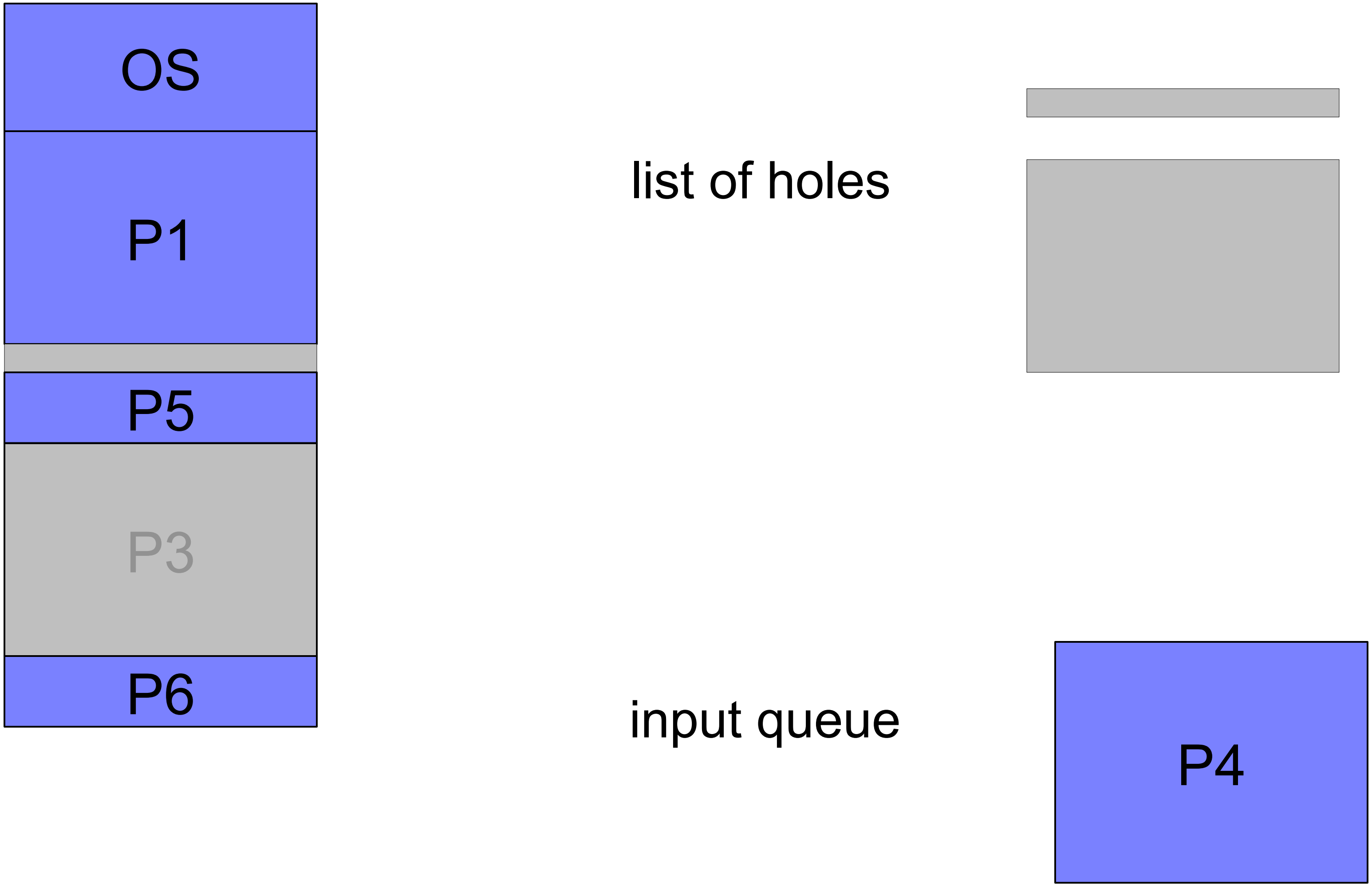
Memory Partitioning



Memory Partitioning



Memory Partitioning



Memory Allocation Algorithm

- Picking the next process:
  - Option #1: First-Come-First-Serve (FCFS)
    - Fast to compute, but may delay small processes
  - Option #2: Allow smaller processes to jump ahead
    - Slower to compute, favors small processes
    - This is what the example showed, and thus P4 was denied access longer than it would have with Option #1
  - Option #3: Something more clever
    - Limit the “jumping ahead”
      - e.g., only 3 processes following process X may jump ahead of it
    - Do some “look-ahead”
      - Wait for >1 new processes before making a (more informed) decision
      - Picking the right amount of time to wait is tricky
    - ...

Memory Allocation Algorithm

- Picking an appropriate hole for a process
- Three common options
  - First Fit: pick the first hole that's big enough
    - fast and easy
  - Best Fit: pick the smallest hole that's big enough
    - slower as it requires sorting
  - Worst Fit: pick the biggest hole
    - slower as it requires sorting
- Once we found the hole, do we put the process at the top or the bottom of it?
  - The middle's likely not a great idea in general

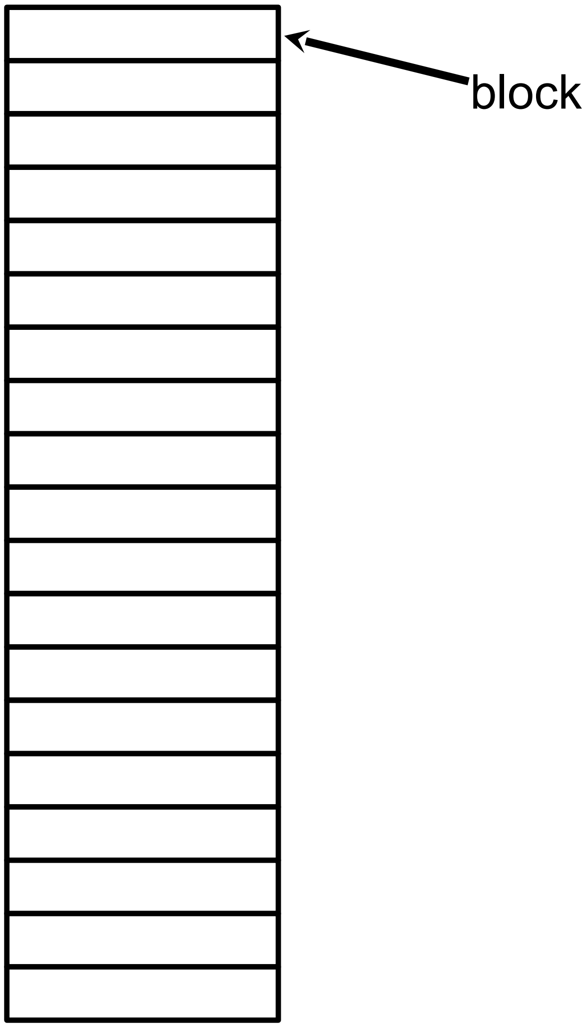
Memory Partitioning

- So what's best?
  - FCFS + First Fit + bottom?
  - Jump Ahead + Worst Fit + top?
- Unfortunately, nothing is best
- These are heuristics to solve an computationally difficult problem
- We can always come up with a scenario for which one combination is better than all the others
  - Even with FCFS + Worst Fit + middle!
- The only thing we can do is run tons of synthetic scenarios, compute averages, and go with what seems best on average
  - Hoping that our scenarios are representative of the real-world
- In this way, we will likely see that "FCFS + Worst Fit + middle" is likely not great, but we cannot prove anything theoretically
  - Or at least nothing useful
- Just like what we saw for scheduling

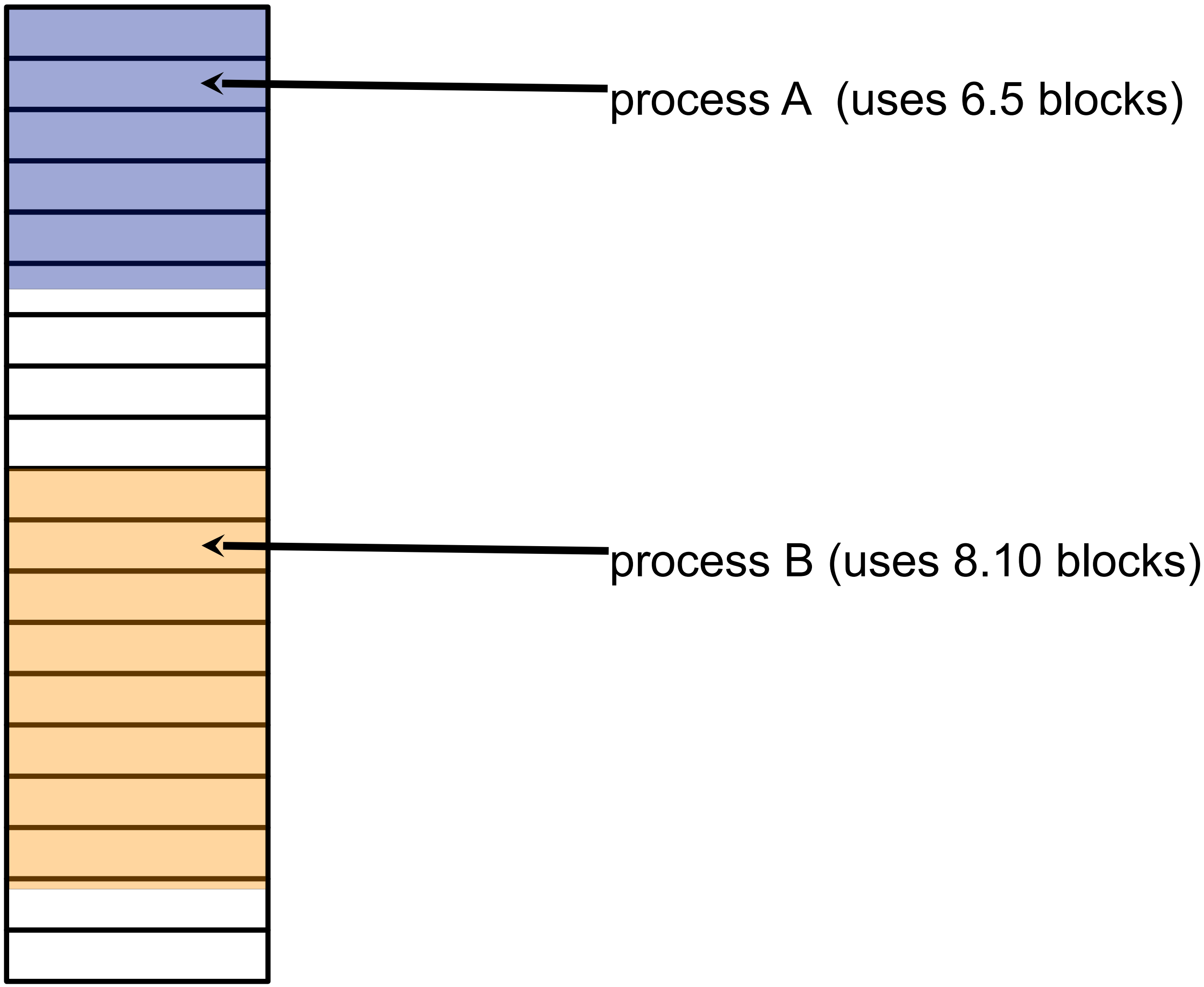
Fragmentation

- Goal: to hold as many processes as possible in memory
- This is very related with minimizing memory fragmentation
  - Fragmentation = number of holes
- For instance, First Fit typically wastes 1/3 of the memory due to fragmentation
  - the “50% rule” (for 2X useful memory, 50% of it is wasted)
- Internal fragmentation
  - We don’t want to keep track of tiny holes
    - Keeping track of a 4-byte hole requires more than 4-bytes of memory in some data structure (e.g., two pointers and an int)!
  - So we allocate memory in multiples of some block size
  - A process many not use all its allocated memory
    - By at most the block size - 1 byte
  - This fragmentation is “invisible” when looking at the list of holes

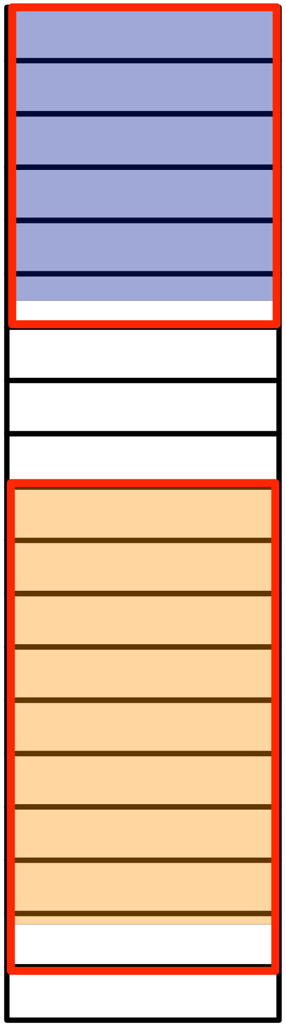
Fragmentation



Fragmentation



Fragmentation



- External fragmentation = 2
  - We have 2 holes
    - one of 3 blocks
    - one of 1 block
- Internal fragmentation = 0.5 + 0.9 = 1.4 blocks
- Tiny blocks: little internal fragmentation, more stuff to keep track of
- Large blocks: large internal fragmentation, less stuff to keep track of

Fragmentation

- One way to deal with fragmentation is compaction
  - What you do when you defrag your hard drive
- This amounts to shuffling memory content
  - Do a memory copy
  - Update the relocation register of the process you moved
    - Only possible with dynamic address binding
- Problems:
  - It's slow (memory copies are slooooow)
  - Problems if processes are in the middle of doing I/O problems (e.g., DMA), just like with swapping

So, where are we?

- Fragmentation is bad
  - e.g., a process of size X may be stuck even though there are hundreds of holes of size X/2
- Shuffling processes around to defrag the memory is expensive and comes with its own problems
- So we cannot reduce fragmentation
- Seems that we're in a bind
- Contiguous memory allocation is just too difficult of a problem to solve well

- We have to do something radically different...