Swapping

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

Swapping

- What if we want to start a new process that would not fit in memory?
- We must save the address space of one (or more) processes from RAM to a “backing store” (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**
  - No problem: programs are relocatable and addresses are virtualized!

- With swapping a process can “be in RAM” or “be on Disk”
- Therefore, a context-switch can involve the disk!!
  - Goes from being lightning fast to being sloth-like slow

Swapping and DMA

- With swapping, a process can be kicked out from RAM to disk by the OS at any time
- This raises a concern with Direct Memory Access (DMA)
  - Reminder: with DMA a process says to the system “while I am doing other things please have the memory system do some memory copy without my involvement”
- Consider a process that has initiated a DMA operation and is swapped to disk
- The DMA controller may have no idea and happily continue to write data (into some other process’ address space, which has replaced that of the one that was swapped out!)

- Operating systems must deal with this (because DMA is so useful we can’t leave without it)
- One option could be: never swap a process engaged in DMA
- In fact, OSes do something else (“paging”, see next Module)

The Bad News about Swapping

- The disk is slooooooooow (even if it's an SSD)
  - e.g., Assume a 5 GiB process address space, a top-of-the-line SSD with 10GB/sec bandwidth: loading a process takes 0.5 seconds
  - This is an eternity from the perspective of the CPU!
- Several ways to cope with slow disks have been used:
  - An OS could swap in/out only processes with small address space (rather than processes with large address space)
  - One can dedicate a disk/partition to swapping (so as to minimize disk seeks on a hard drive)
- One approach is to just not swap, and swapping should be exceptional
  - In older OSes swapping was user-directed (e.g., Windows 3.1)
- Swapping is now often disabled (e.g., on laptops)
  - If the normal mode of operation of the system requires frequent swapping, the system is in trouble (buy more RAM!)
  - But perhaps it's just a temporary rare load spike?
- The real solution is to not swap whole address spaces, but we leave that for the next module

Where are we?

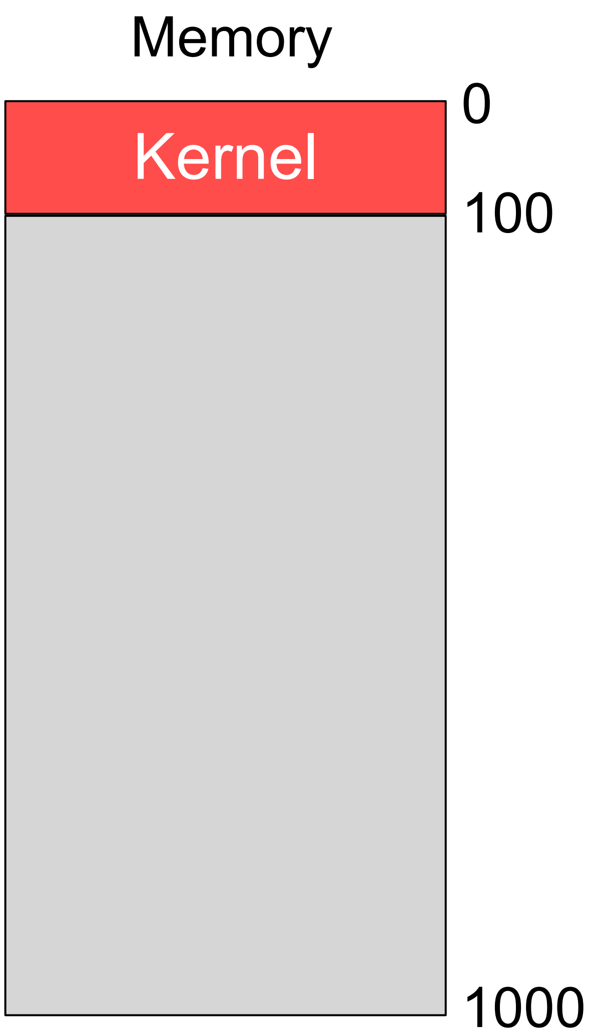
- We now have the mechanisms we need:
  - We know how to give each process a “slab” of memory that can fit anywhere in RAM (address virtualization)
    - Or one slab per segment
  - We know how to swap processes in and out of memory
- We now need a policy to decide how to place each slab in memory:
  - We want to have as many process address spaces in memory as possible
  - We want to minimize swapping
- **Key Question:** What is a good policy?

Memory Allocation

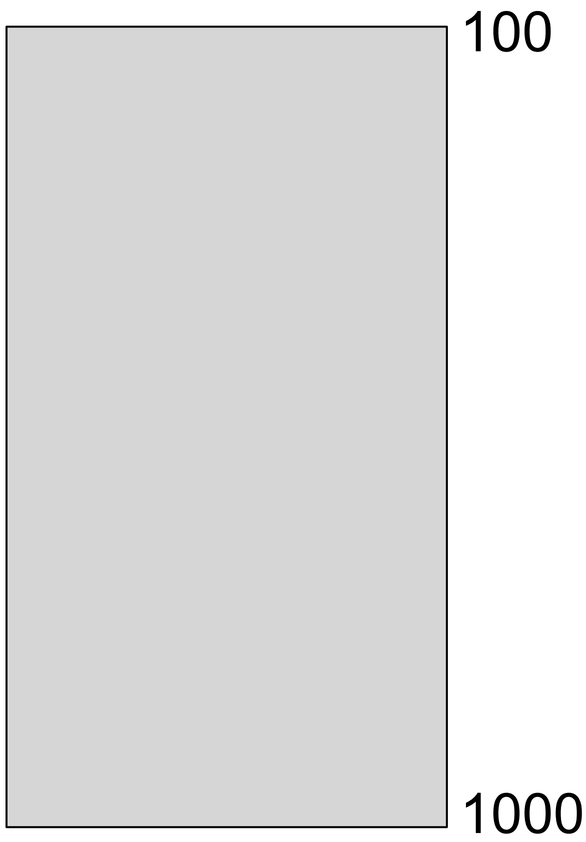
- The kernel must keep a list of available memory regions or “holes”
- When a process arrives, before scheduling it, it is placed in a “I need memory” input queue
- The kernel must make decisions:
  - Process Selection: Pick a process from the input queue
  - Memory Allocation: Pick a hole in which the process will be placed (and update the list of holes)
- Then, the kernel can just place the process’ PCB into the ready Queue

- This problem is known as the Dynamic Storage Allocation Problem
- It’s an on-line problem (we don’t know the future)
  - As opposed to off-line (we know the future)
- **Objective:** Hold as many processes in RAM as possible

Memory Allocation Example

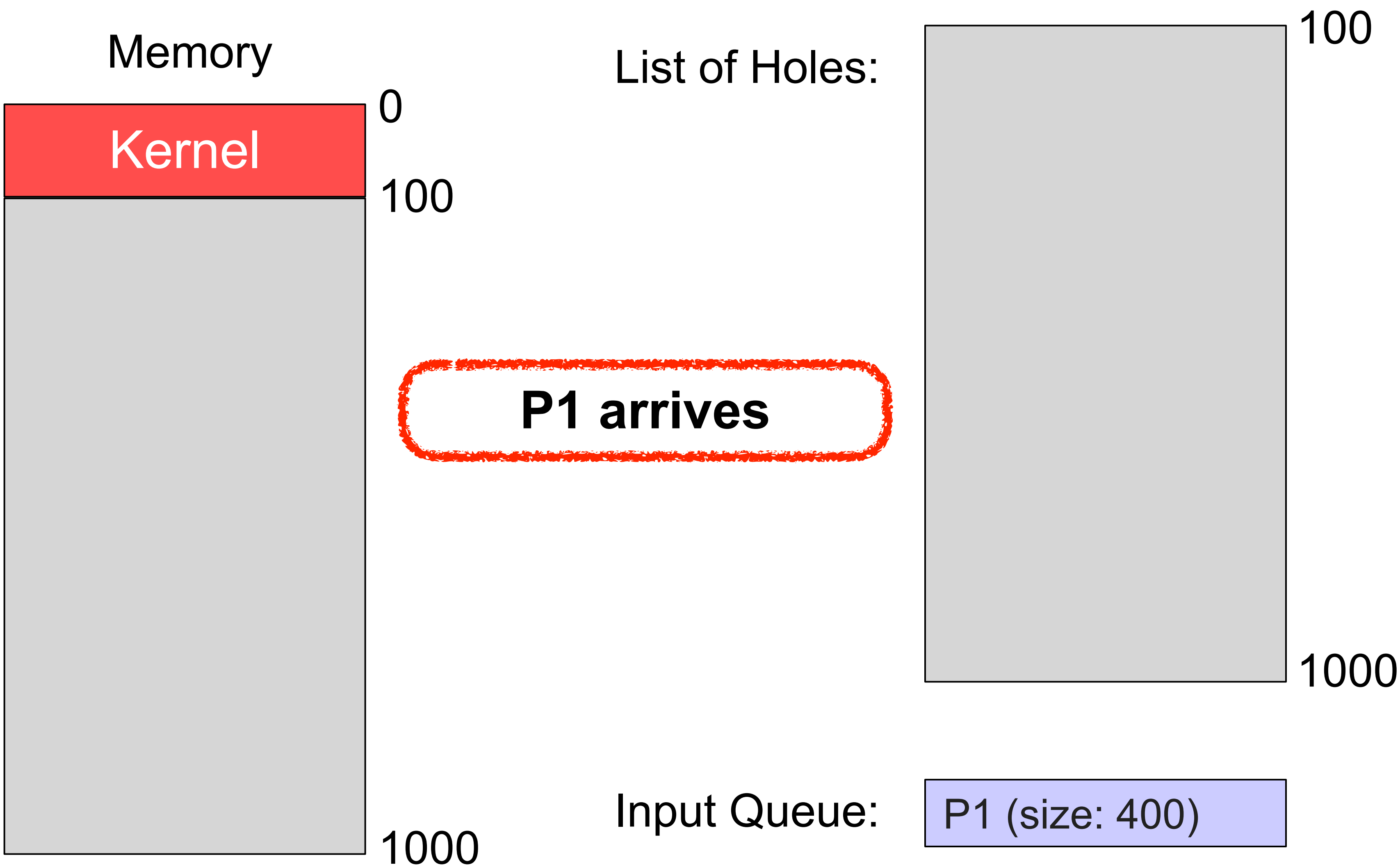


List of Holes:

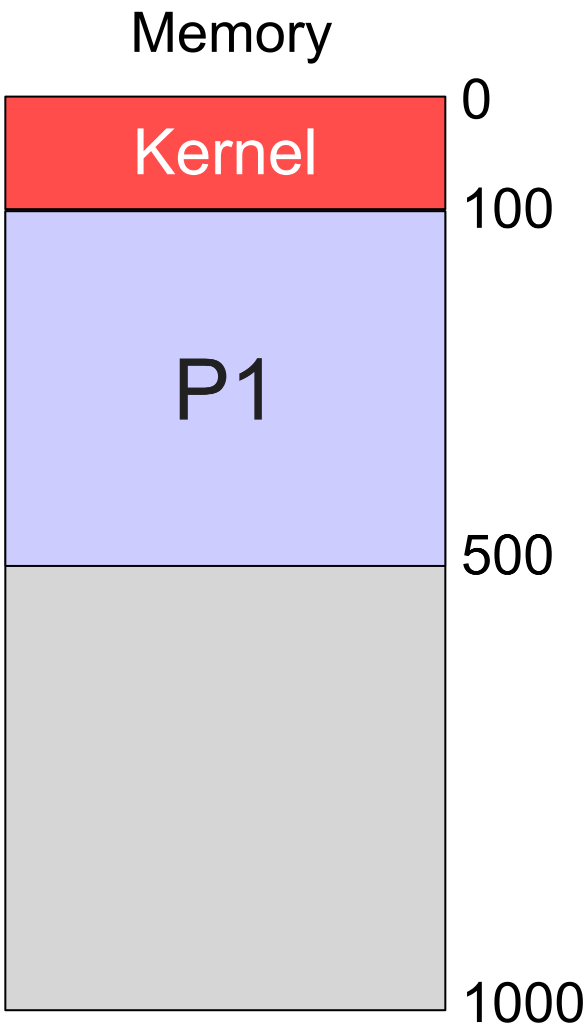


Input Queue: ∅

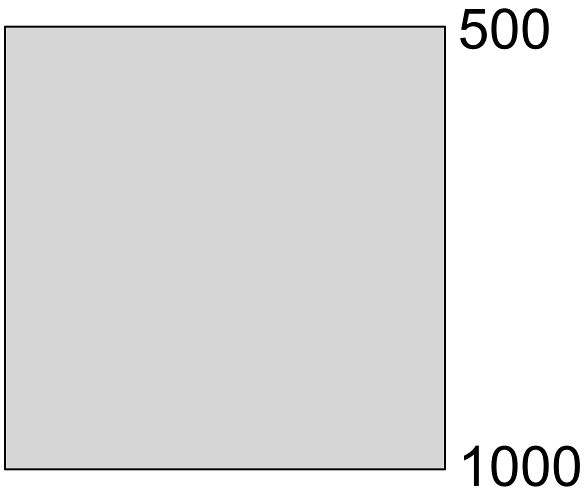
Memory Allocation Example



Memory Allocation Example



List of Holes:

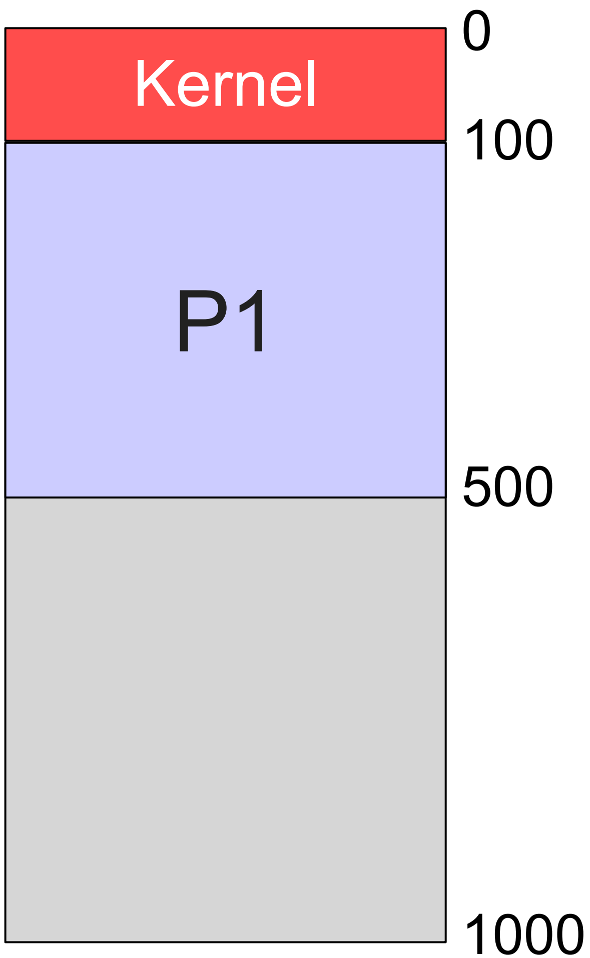


P1 is loaded

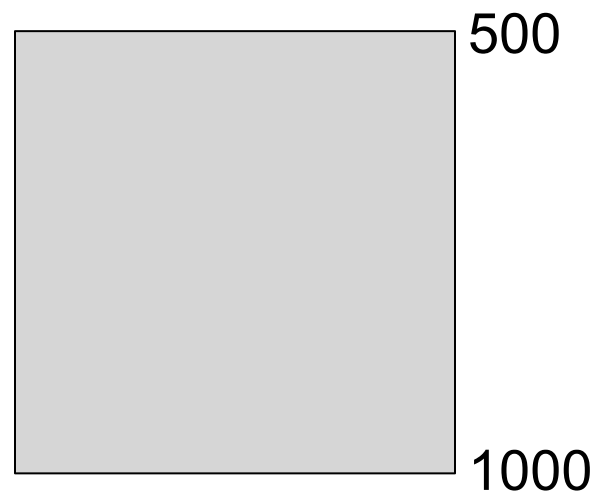
Input Queue: $\emptyset$

Memory Allocation Example

Memory



List of Holes:

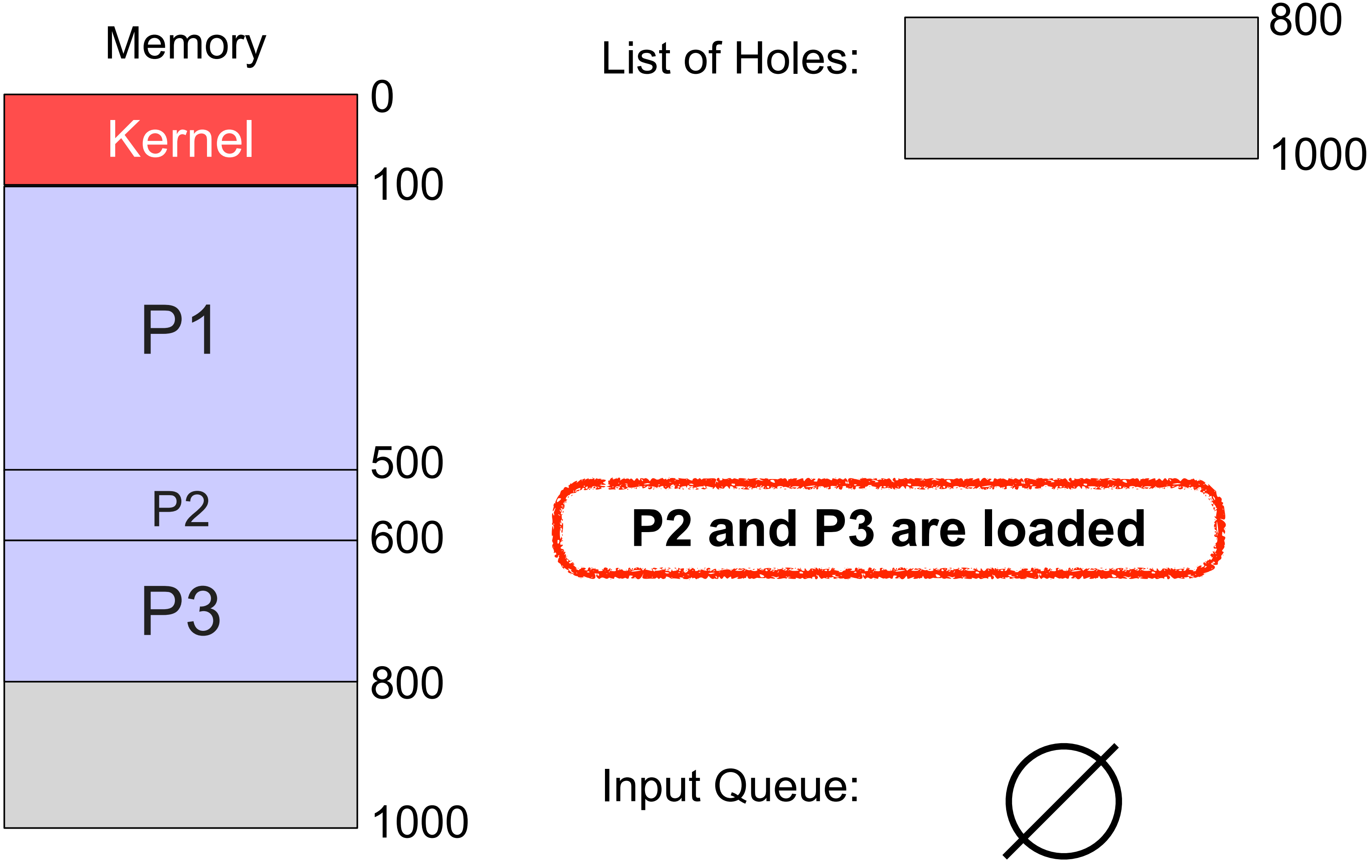


P2 and P3 arrive

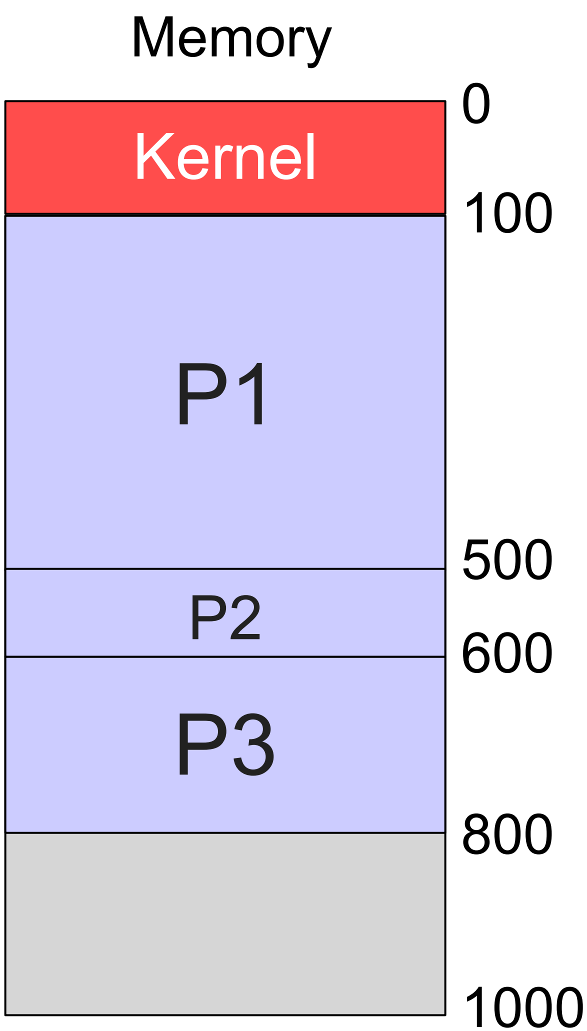
Input Queue:

- P2 (size: 100)
- P3 (size: 200)

Memory Allocation Example



Memory Allocation Example

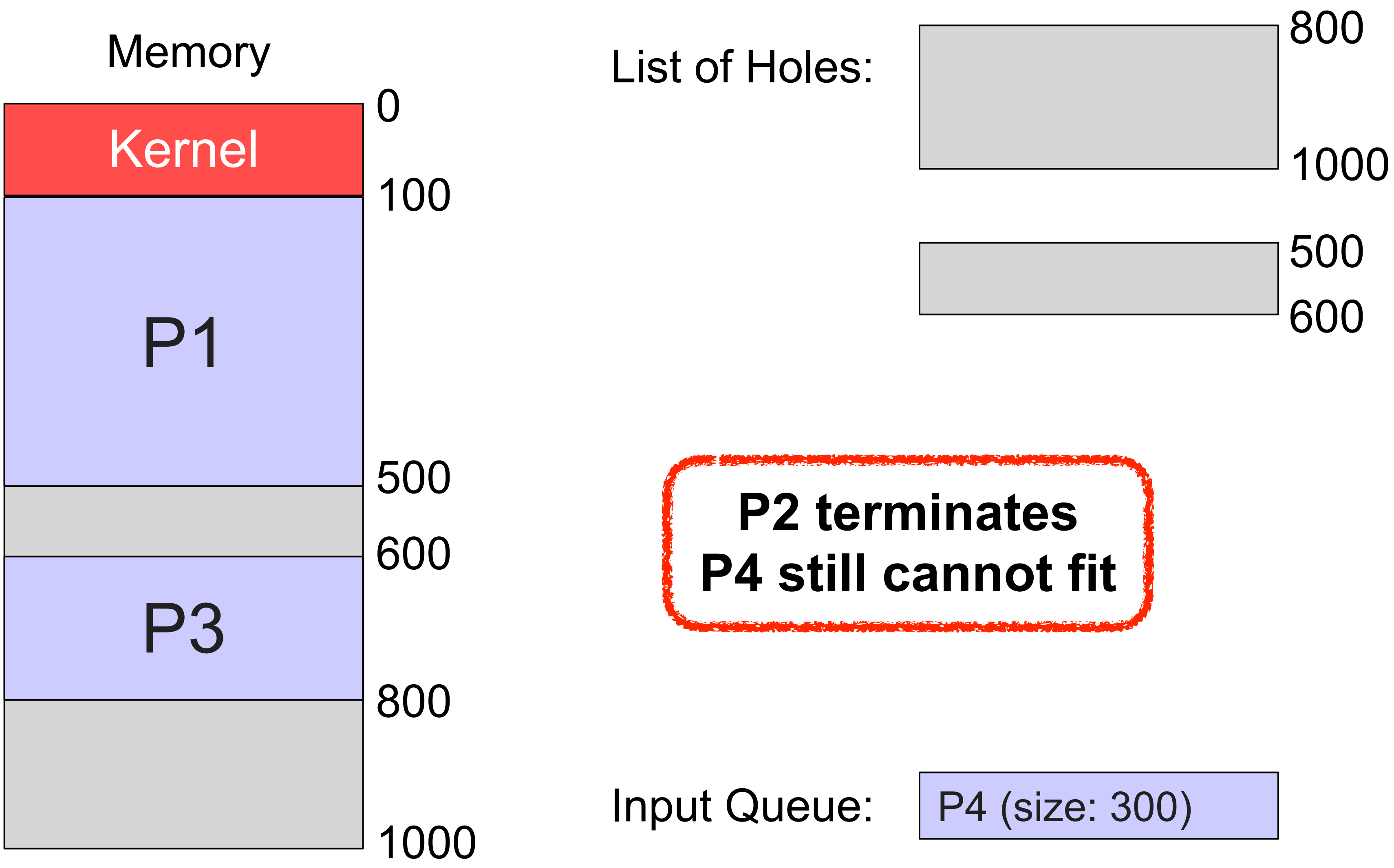


List of Holes:

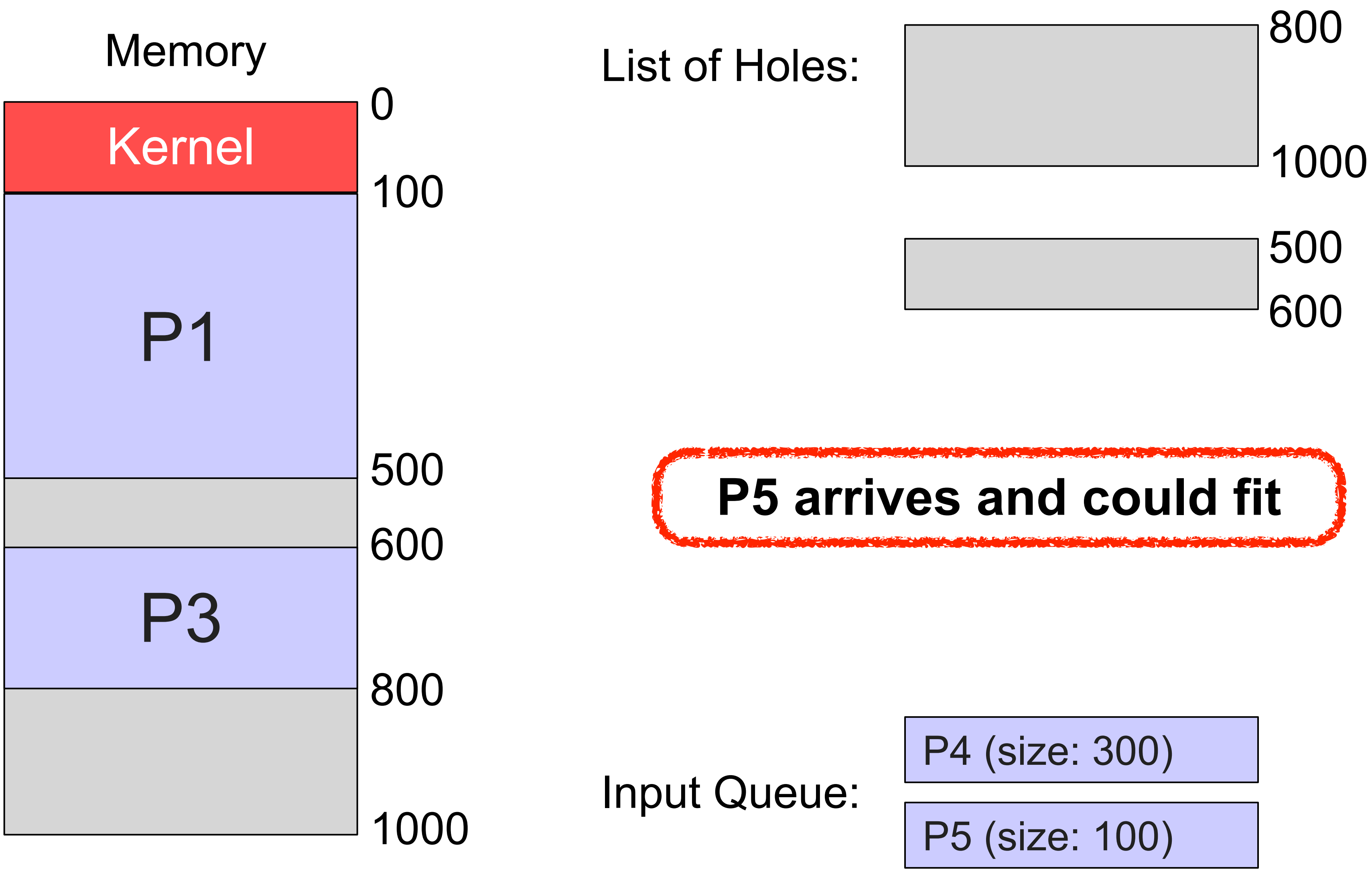
P4 arrives, and can't fit

Input Queue: P4 (size: 300)

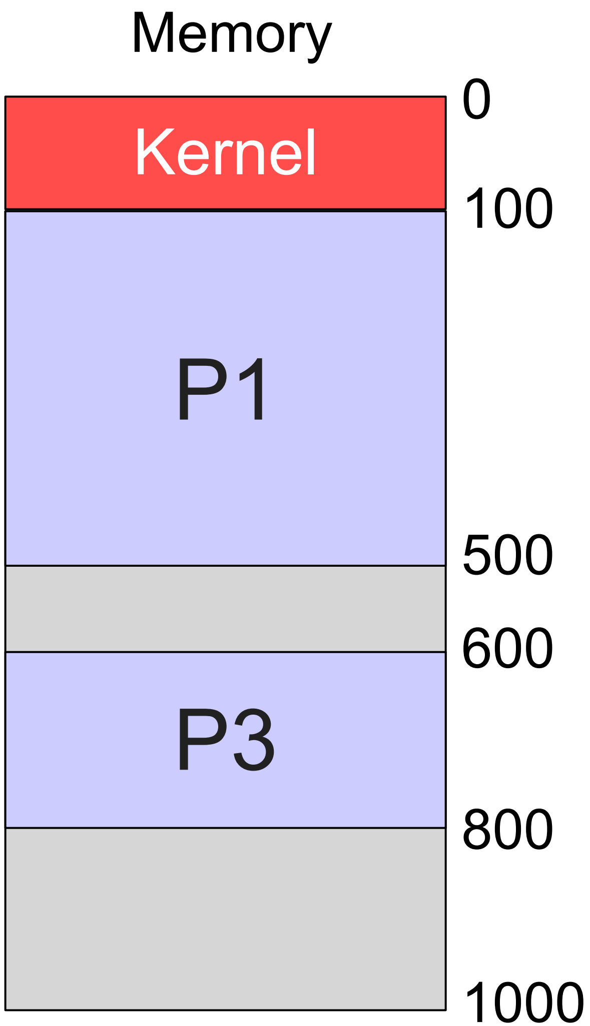
Memory Allocation Example

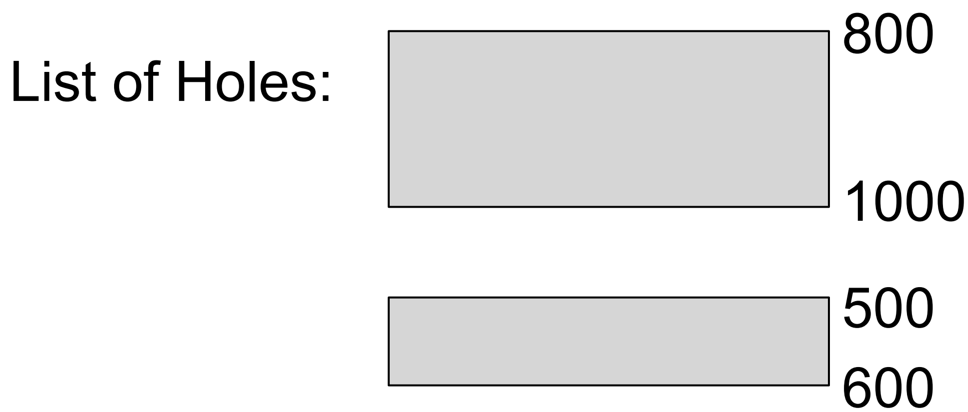


Memory Allocation Example

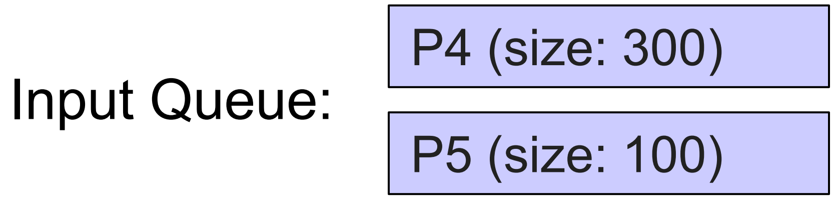


Memory Allocation Example





- There is enough space for P4, but we have **fragmentation**
- Tempting to pick P5 first?
- It's a Scheduling problem



Memory Allocation Strategies

- **Question 1/3:** Which process should be picked?
- First Come First Serve?
  - Easy, fast to compute, may delay small processes
  - Once again, the supermarket shopping analogy
- Allow smaller processes to jump ahead?
  - Slower to compute, favors small processes
- Something more clever?
  - Limit the “jumping ahead” (e.g., you cannot jump over more than 3 processes)
  - Look ahead (e.g., instead of making a decision right now, wait for a few more processes to arrive to get a clearer picture of what the workload looks like)
- ...

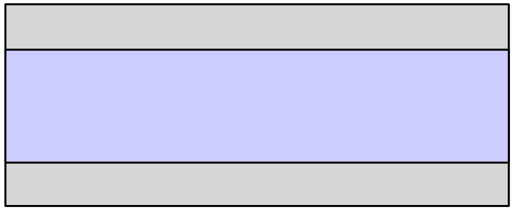
Memory Allocation Strategies

- **Question 2/3:** Which hole should be picked for the process that was picked?
- First Fit?
  - Pick the first hole that is big enough
- Best Fit?
  - Pick the smallest hole that is big enough
- Worst Fit?
  - Pick the biggest hole

Memory Allocation Strategies

- **Question 3/3:** How should the picked process be placed in the picked hole?
- Top?

- Middle?



- Bottom?

Memory Allocation

- What should we do?
  - FCFS + First Fit + Top?
  - Jump Ahead + Worst Fit + Bottom?
- We are trying to solve an on-line (don't know the future) bin-packing (fit boxes in bins) dynamic (boxes can disappear) problem: this is hard!
  - In fact it's NP-hard even if we know the future!
- The above combinations are heuristics that hopefully produce decent solutions in practice
- We can always come up with a scenario for which one combination is better than all the others
  - Even for the seemingly "stupid" FCFS + Worst Fit + Middle
- This is in essence the same story as for CPU scheduling

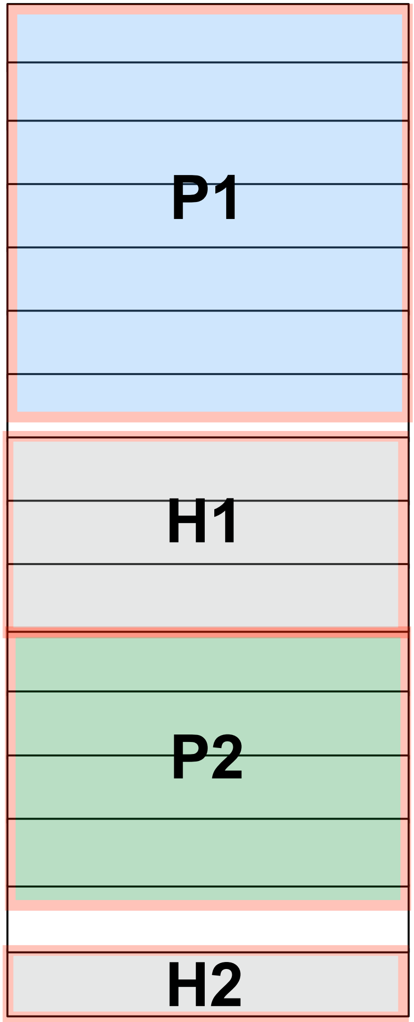
External Fragmentation

- Recall our objective: hold as many processes as possible in memory
- What makes it difficult is external fragmentation
- We have already seen fragmentation on an example
  - There were two small *disjoint* holes that together would have been big enough to accommodate a process
- The external fragmentation is defined as the number of holes
- For a given amount of available RAM, we're always happier with a single large hole than with several smaller holes
  - Just like you'd rather have a chest of drawers with 4 large drawers than with 100 tiny drawers that have the same total volume but can each only fit tiny things
- But, because processes terminate whenever they want to, we cannot avoid external fragmentation
- What about compaction?
  - Just like defragging a hard drive
  - But moving processes around means a lot of slow memory copies
  - And it creates complicated issues with I/O, DMA, etc.
  - So no OS does it

Internal Fragmentation

- Do we want to keep track of tiny holes?
  - The list of holes in the kernel is a list of data structures
  - Each data structure has: (i) a base address and (ii) a size
  - On a 64-bit architecture, this data structure would be 16 bytes
  - Plus the pointer to it, we have 24 bytes
  - So, I don't want to use 24 bytes to keep track of, say, a 16-byte hole!
- In practice, an OS would allocate slabs that are multiples of some "block size" (e.g., a number of KiB)
- Downside: a process may then not use the whole slab and some space is wasted
- This is called internal fragmentation

Fragmentation Example (1KiB Blocks)



- Process P1 uses 6.8 KiB out of 7
- Process P2 uses 4.3 KiB out of 5 1-KiB blocks
- **External fragmentation:** 2 holes:
  - H1: 3 KiB
  - H2: 1 KiB
- **Internal fragmentation:** (1-0.8) + (1-0.3) = 0.2 + 0.7 = 0.9 KiB
- Smaller blocks? lower internal fragmentation, but more blocks to keep track of
- Larger blocks? higher internal fragmentation, but fewer blocks to keep track of

Main Takeaway

- Our objective was to allocate a contiguous slab of memory to each process (or to each process segment) so that their address spaces can be in RAM
- The mechanisms are “easy”
  - Relocatable code with virtualized addresses
  - Swapping processes in and out
- But finding a good policy is hard
  - FirstFit, BestFit, WorstFit are three classic algorithms
- Internal and External Fragmentation

Conclusion

- Our objective was to allocate a contiguous slab of memory to each process (or to each process segment) so that their address spaces can be in RAM
- The mechanisms are “easy”
  - Relocatable code with virtualized addresses
  - Swapping processes in and out
- But finding a good policy is really hard
  - For process picking, hole picking, placement in hole
- It’s hard because fragmentation is unavoidable and wastes RAM

- One way to make it less hard is to try to have small address spaces, which we discuss in our next set of lecture notes...

- Before, let’s look at the Sample Homework Assignment
- And look at some Practice Problems…