DSC-204A: Scalable Data Systems, Winter 2024

Basics: OS-3

Lecturer: Hao Zhang *Scribe: Tanmay Pardhi, Tejas Ramesh*

1 Recap Practice:

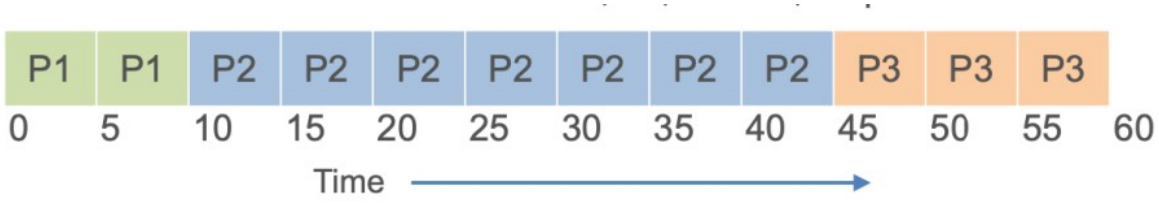


Figure 1: P1, P2, and P3 are of lengths 10, 35, and 15 units resp. and arrive at times 0, 10, and 20 resp.

Here is a Gantt Chart for 3 processes of the given lengths that arrive at the different times given.

A) What is the rough average response time?

B) What is the rough average turnaround time?

C) Which scheduling policy/policies discussed in class (FIFO, SJF, SCTF, RR) may produce this given schedule? Explain clearly.

Solution:

A) Response Time = Start Time — Arrival Time

For Process P1, Response time = 0 - 0 = 0

For process P2, Response time = 10 - 10 = 0

For process P3, Response time = 45 - 20 = 25

Average Response time = $\frac{25}{3}$ = 8.33

B) Turnaround Time = Completion Time — Arrival Time

For Process P1, Turnaround time = 10 - 0 = 10

For Process P2, Turnaround time = 45 - 10 = 30

For Process P3, Turnaround time = 60 - 20 = 40

Average Turnaround time = $\frac{80}{3}$ = 26.67

C) The scheduling policy used here is FIFO, as the process that arrived first is completed first.

SJF cannot be the answer because in SJF, P3 would be prioritized before P2 since P3 has a shorter job length in comparison to P2.

RR cannot be the answer because there is no context switch used in the scheduling.

SCTF cannot be the answer because the completion time of P3 is less than P2 and when P3 arrives P2 will still have 25 units of time left so P3 should have been scheduled before P2.

2 Review Preemption case: SCTF

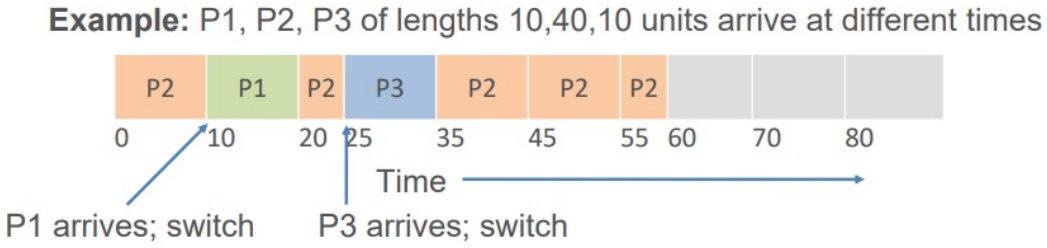


Figure 2: Preemption

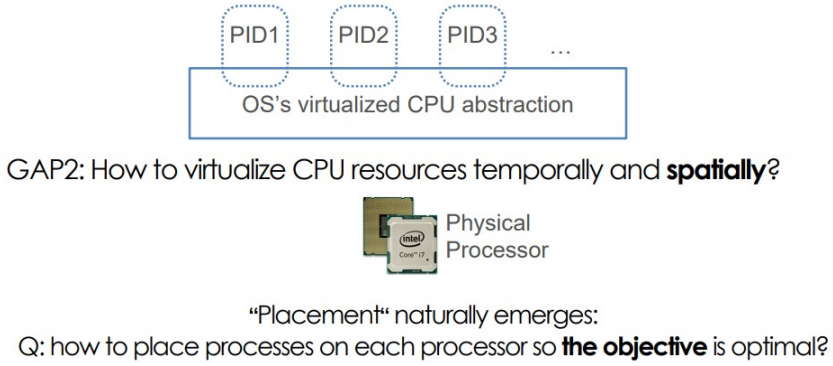
Preemption is similar to context switch where we will swap the current job to backup and process the other job instead.

We have a schedule where we have started working on P2 at time 0 and P1 arrives at time 10. Since the scheduling policy being used is Shortest Completion Time First (SCTF) we want to preempt P1 before P2.

In this example when P1 arrives we preempt the process P2 and process P1 instead and when P1 gets finished we again work on the process P2.

3 Implementing Scheduling:

GAP 2: How to virtualize CPU resources temporally and **spatially**?



We have seen how we can virtualize the CPU temporally and now we want to see how we can virtualize the CPU spatially

Modern CPU has many many cores and in Cloud computing, we also have many many nodes.

We need to decide how to place these jobs/processes spatially. Which core the process should be placed on?

We need to do it such that the objective is optimal. The objective can be utilization of cores or completion time of the processes.

A key component while optimizing the placement is Load Balancing. It ensures all the cores are equally utilized otherwise one core will get overloaded in comparison to other and CPU/GPU cycles will be wanted on the idle core.

3.1 Placement in Deep Learning:

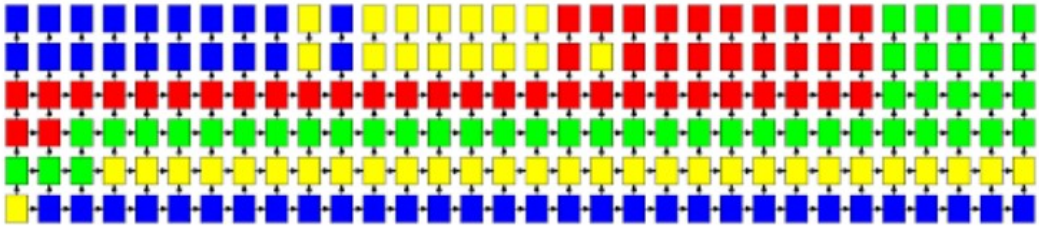


Figure 3: Deep Neural Network

Since this is a very big neural network we cannot finish the job with 1 CPU/GPU/node. So we need to divide this neural network into many parts, here each block represents one part. We want to place this neural network onto those GPUs such that the job gets finished in the shortest amount of time. Each color here represents a core. This placement of the parts needs to be calculated using algorithms that maximize the objective. Due to the large number of devices and nodes, it becomes exponentially difficult. In Deep learning, people need to solve this to make their Neural Network train faster and the way they solve it is they either design some objective and do some observations and maths or they just keep trying to do some placement to get the optimal results, do modifications and keep trying with multiple placements until ultimately they find the right placement. For this another approach is to use the machine learning method, Reinforcement learning. With every placement, a trajectory is drawn and we try to optimize this using reinforcement learning.

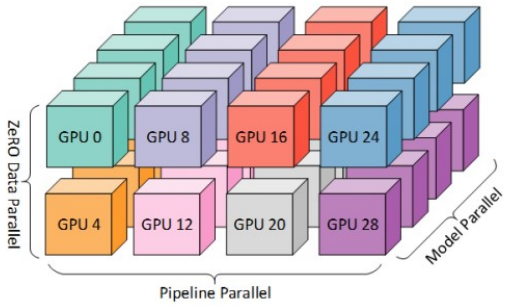


Figure 4: Deep Neural Network

This is a very effective placement used to train GPT-3. Here we divide the neural network into many

spatial blocks. We put each block into a GPU and this block is connected spatially to the blocks near them. However, the leftmost block is not connected to the rightmost block. We optimize this kind of placement to make sure we can deploy GPT3 on huge cloud computing clusters that are interconnected. As we can see from the top the Neural Network is deeper while from the bottom it is more regular.

3.2 Last issue in Placement: Inter-process communication (IPC)

In multicore environments, we place different processes in different CPU cores/GPUs to make sure you optimize the objective. In process management, we need two processes to communicate. When they are in one OS the communication can be handled by a concept called Inter-Process Communication (IPC) managed by OS. Inter-process communication is provided in System Calls API.

4 Memory Management

4.1 Recap

We evolved from the Strawman solution to the spatial-temporal sharing/scheduling. We made a simplified assumption that we assign one-third of the total memory to each APP. Let's repeat our questions now:

- **Question**: Can I run N apps with total memory > physical memory cap?
  Answer: We cannot; because we did a fixed allocating scheme.

Apparently, this solution is very inefficient. We are going to redesign scheduling and memory management and tell you why memory management has become what it looks like today. Let's see whether we can find a better solution.

We put all three processes in the memory pool, and they get what they need. For example, the first process, if its memory consumption is not super big, can just take fewer than one-third, and the rest of the memory can be allocated to any process that needs more memory. How exactly we do this sharing is pretty complex because now we have access to the entire memory addresses, and they can interfere with each other. What if two processes ask for the same amount of memory, and it is greater than some amount that the memory system cannot allocate? How do we handle that?

4.2 Memory Management v0

Let's develop a v0 memory manager. One way is to organize it into many blocks, and we just put contents in these blocks. Now we have three different processes P0, P1, and P2, and we need to start working on these processes, and the operating system needs to allocate memory to these processes. We just give them the amount of memory they need.

- **Problem**: There's a gap between P0 and P1 and another gap after P2. This gap happens because of the way we manage the blocks. This concept is called **internal fragmentation**. Internal fragmentation happens when we want to allocate memory for the process, but we allocate a little bit more because of the formatting of this memory space. This space is wasted because we cannot use it for any other processes.

  Solution: So how to solve this? We reformat the disc, into smaller blocks. As long as the block is small enough, we will not have internal fragmentation.

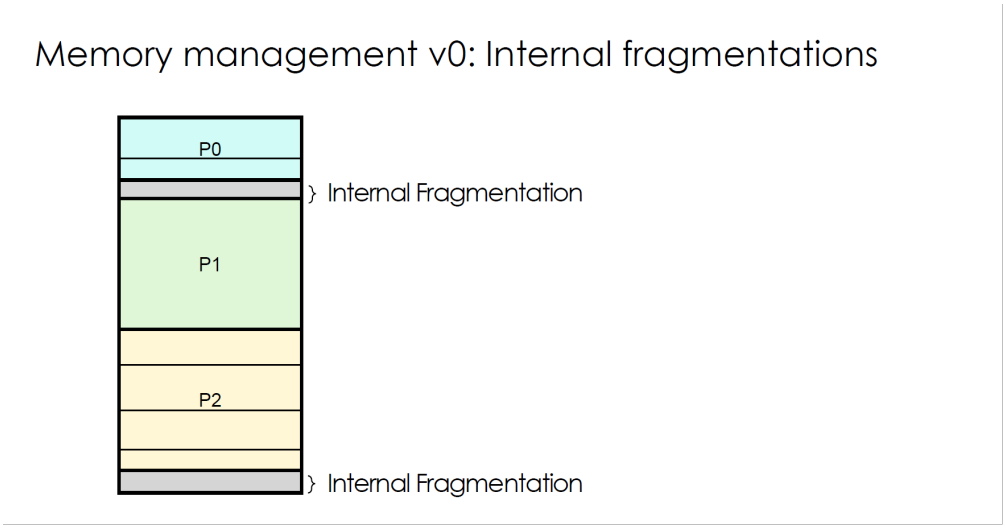


Figure 5: Memory Management v0

- **Question**: What is the maximum possible amount of internal fragmentation per process? **Answer**: The internal fragmentation per process will not exceed one block.

4.3 Memory Management v1

Now we have a memory management v1. Let's address another issue.

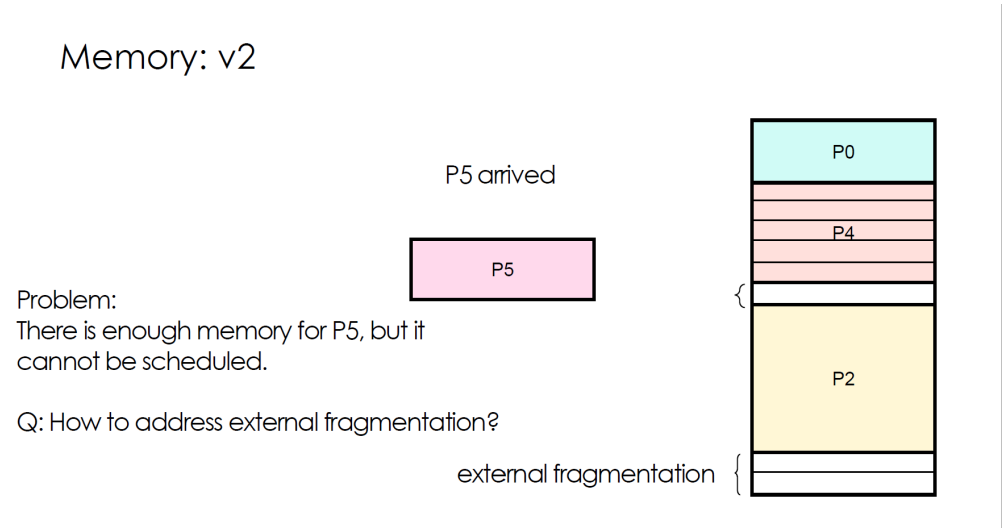


Figure 6: Memory Management v2

- **Problem**: Let's assume that P1 finishes. Meanwhile, P4 arrives. We want to request memory for P4 and find that the requirement for P4 is smaller than the original space allocated for P1. But what happens next when P5 arrives? There is enough total memory that can support P5, but we cannot

support it because those two blocks are not continuous. This is called **external fragmentation**. Internal fragmentation is when you allocate something bigger than a process needs, whereas external fragmentation occurs when you have enough memory, but because that memory is not continuous, you experience another form of fragmentation. This is external fragmentation. 20 years ago, Windows was like this. If you run it for a few days, it becomes super slow.

- **Question**: How to address the external fragmentation?
  Answer: There's no good way. It is still an area of study.

There are problems with this approach:

1. **Problem**: Even if we divide the memory space into blocks, we still cannot schedule any process with a total memory consumption greater than the memory cap.
2. **Problem**: We don't know how much memory space is required by each process. For example, when you open a Word document, you keep typing more content. When implementing a program that is reading or writing something, it keeps generating things. There is a problem with this approach; for example, P1 might eventually only use a small amount of memory, but we overallocated it at the beginning. The extra space called P1 reserve will be wasted.
3. **Problem**: What if we know exactly how much memory each process will eventually use? Even if we know this, it is still not good because the memory consumption of each process dynamically grows. We are still facing the same problem of overallocation.

In summary, we have internal fragmentation that can be addressed by reformatting the memory space into smaller blocks. We have external fragmentation, which is still a problem today. And, we have reservation overhead, which happens regardless of whether we know the total amount of individual memory that will be used by each process.

4.4 Virtual Address Table

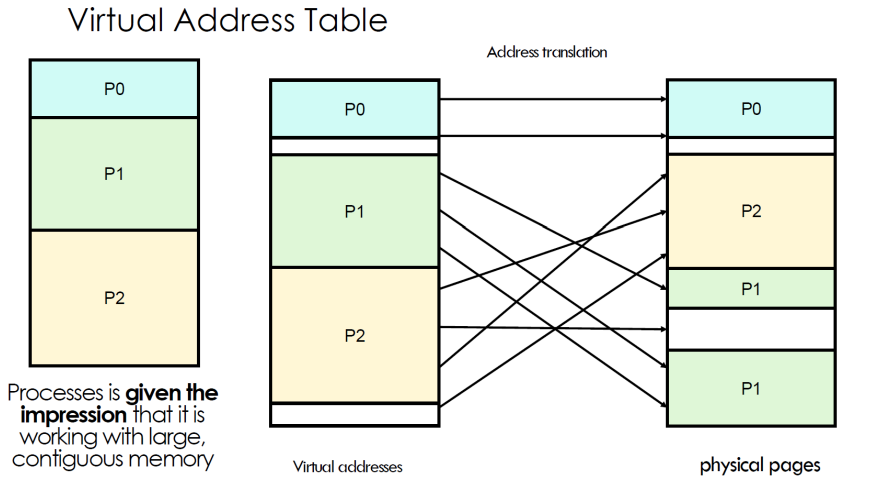


Figure 7: Virtual Address Table

So with this idea, we came up with a solution called a virtual address table. We do not let the process directly interact with physical memory, but we will build an equivalent to the physical memory. This one is not a real memory; there's no space weight. We just keep a table that maps each block to its address. So it does not take up any space. But what we do is ask the computer to directly interact with the virtual address. We map a block from the virtual address to the physical address. When a process asks for memory, it first interacts with virtual addresses, and the address translation will eventually compile the virtual address into a physical address. One apparent advantage is that this process is given the impression that it is working with large, near-infinite, and continuous memory.

How? We create a very large virtual address, which is larger than the physical one, but we map different pages in the virtual address to the same pages in the physical address. The process will only see the virtual addresses. Second, why is it always continuous? Because of mapping; we always put processes continuously into the virtual addresses. But the virtual addresses, like many blocks, although they are continuous in this virtual table, can be mapped into non-continuous physical tables.

Some important concepts:

1. **Pages**: Pages are those blocks. It's an abstraction of fixed-sized chunks of memory storage. People make it for 4KB or 16KB, which means that our internal fragmentation will be bounded by this size. OS memory management identifies pages uniquely and reads/writes pages from/to disk when requested by a process.
2. **Page Frame**: Virtual slot in DRAM to hold a page's content
3. **Virtual Memory**: The operating system memory mechanism contains two addresses: virtual address and physical address. And the address translation has no overhead today. The OS maintains a free space list to tell which chunks of DRAM are available for a new process and avoids conflicts.

This virtual table addresses the problems we discussed:

1. **Problem**: In the physical memory allocation scheme, we can never schedule processes with memory consumption greater than the memory cap.
   Solution: We can create more virtual addresses than physical addresses. When we find that the physical addresses are full, we just move it down to another tier. The operating system implements this mechanism, which we call **swap**. For example, in machine learning, when you try to train a model where the required memory is bigger than your GPU memory, you swap out all your previous content away and swap something in.
2. **Problem**: What if we are unsure about how much memory each process will eventually use?
   Solution: We reserve memory on the virtual tables and resolve the mapping between virtual and physical pages on the fly.
3. **Problem**: What if we know exactly how much memory the processes will eventually use?
   Solution: Because we do everything on the fly, we minimize the opportunity costs.

4.5 Scheduling in ChatGPT

We're going to connect this to ChatGPT. Each of these requests acts as a process, and they are going to consume memory because you are decoding and generating tokens, and each token will take a certain amount of memory from the GPU.

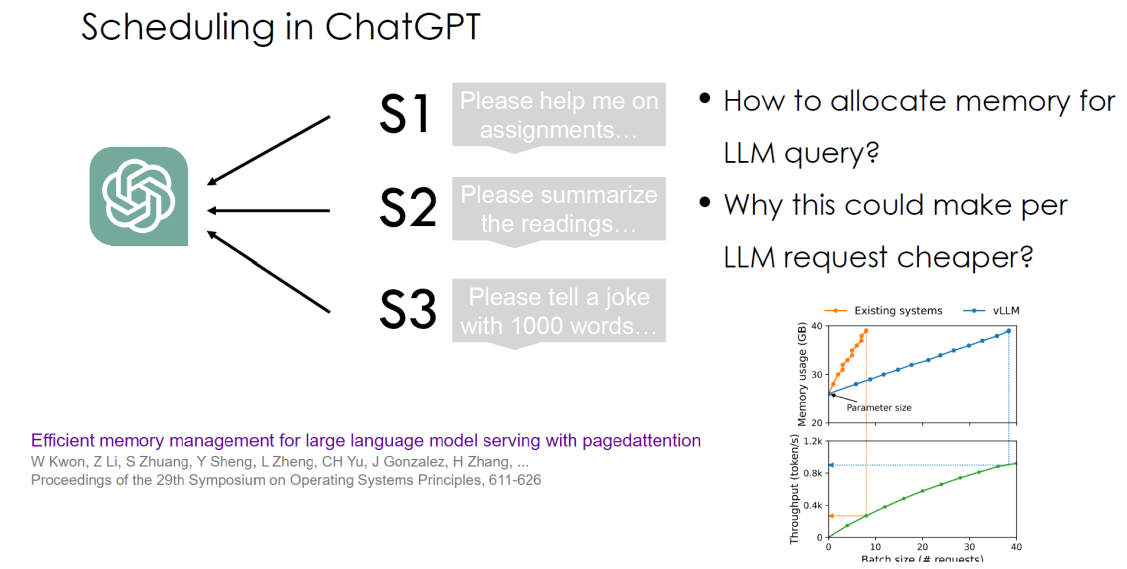


Figure 8: Scheduling in ChatGPT

- **Question**: Do we know the eventual memory consumption of each request?
  Answer: We don't know. You don't know how many tokens will be generated eventually. This is exactly the problem we saw in paging. We don't know the eventual memory consumption of each request, but we know that it will keep growing.
- **Question**: How to allocate memory for these requests?
  Answer: Perform paging and allocate memory on the fly when you decode more tokens. Think of each LLM query as a process and follow the same kind of page memory management.
- **Question**: Why could this make an LLM request cheaper?
  Answer: Because you will use less memory to serve more requests. If you directly interact with physical memory, you might maximum support like ten requests. You will hit a memory cap. But with paging, you can support more requests, and the opportunity cost will be reduced.

The system throughput, which is how many requests you can serve per second, will also improve. That is, your system will become even faster. This is something very specific to GPUs because all these requests are running on GPU computation.

Consider a scenario where you don't use paging; you allocate memory statically per request. Let's say you are given 40 gigabytes of GPU memory, then you can support ten requests and you will hit the memory cap. But if you use paging, we reduce opportunity costs. We can use the same amount of 40 gigabytes of memory to support 100 requests.