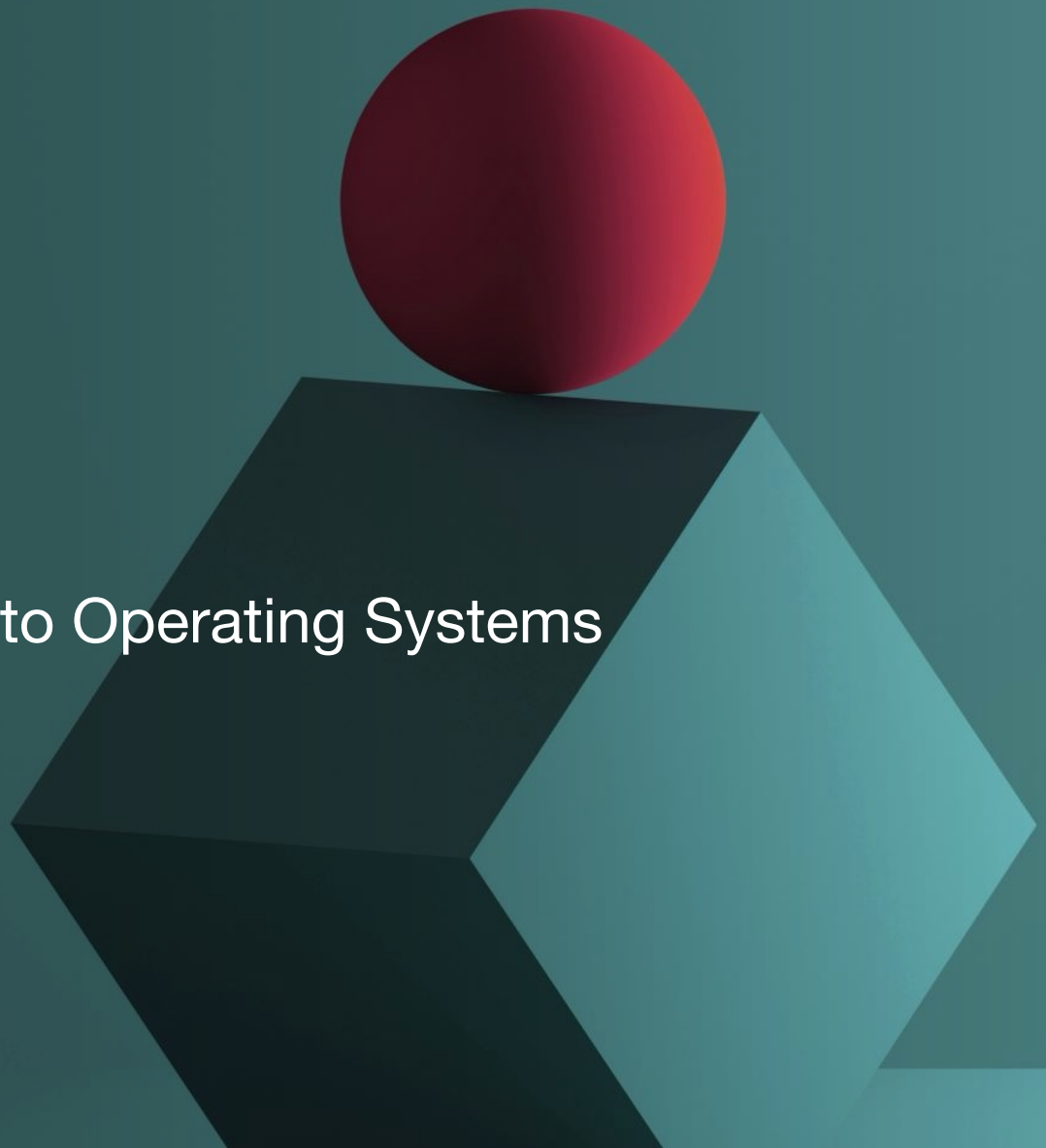
Tutorial 4

CS2106: Introduction to Operating Systems



Tutorial 3 Question 5

The simple MLFQ has a few shortcomings. It is best described by the scheduling behaviour for the following two cases:

a) **(Change of heart)** A process with a lengthy CPU-intensive phase followed by I/O-intensive phase.

b) **(Gaming the system)** A process repeatedly gives up CPU just before the time quantum lapses.

MLFQ Rules

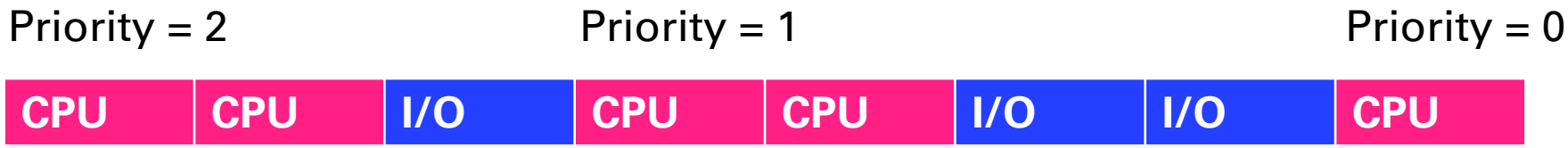
- **Basic rules:**
  1. If Priority(A) > Priority(B) ➔ A runs
  2. If Priority(A) == Priority(B) ➔ A and B run in RR

- **Priority Setting/Changing rules:**
  1. New job ➔ Highest priority
  2. If a job fully utilized its time slice ➔ Priority Reduced
  3. If a job give up / blocks before it finishes the time slice ➔ priority retained.

Tutorial 3 Question 5

Case A: (Change of heart) A process with a lengthy CPU-intensive phase followed by I/O-intensive phase.

- The process can sink to the lowest priority during the CPU intensive phase.
- With the low priority, the process may not receive CPU time in a timely fashion during the I/O phase which degrades the responsiveness.



Time Quantum = 2 TU
One Box = 1 TU

Tutorial 3 Question 5

Case B: (Gaming the system) A process repeatedly gives up CPU just before the time quantum lapses.

- The process will retain its priority.
- Since all process enter the system with the highest priority, a process can keep its high priority indefinitely by using this trick and receive disproportionately more CPU time than other processes.

Tutorial 3 Question 5

The following are two simple tweaks. For each of the rules, identify which case (a or b above) it is designed to solve, then briefly describe the new scheduling behaviour.

i. (Rule 1 – Accounting matters) The CPU usage of a process is now accumulated across time quanta. Once the CPU usage exceeds a single time quantum, the priority of the task will be decremented.

ii. (Rule 2 – Timely boost) All processes in the system will be moved to the highest priority level periodically.

Tutorial 3 Question 5

(Rule 1 – Accounting matters) The CPU usage of a process is now accumulated across time quanta. Once the CPU usage exceeds a single time quantum, the priority of the task will be decremented.

- Rule 1 is designed to solve case B so that a process that repeatedly gives up CPU will not always have the highest priority.
- By accumulating CPU usage, processes in Case B will eventually overshoot a single time quantum.
- This prevents such processes from hogging the CPU

Tutorial 3 Question 5

(Rule 2 – Timely boost) All processes in the system will be moved to the highest priority level periodically.

Rule 2 is designed to solve case A, so that by periodically boosting the priority of processes, a process with different behaviour phases may get a chance to receive CPU time and remain responsive even after it has sank to the lowest priority.

Question 1

It’s Kahoot Time!

Question 1(a)

Under what conditions does FCFS (FIFO) scheduling result in the shortest possible average response time?

Time b/w request and response (i.e. when a process gets scheduled to run on CPU)

$$\text{Avg Response Time} = \frac{0+1+4}{3} = \frac{5}{3} = 1.67$$

$$\text{Avg Response Time} = \frac{0+5+8}{3} = \frac{13}{3} = 4.33$$

$$\text{Avg Response Time} = \frac{0+3+6}{3} = 3$$

Question 1(a)

Under what conditions does FCFS (FIFO) scheduling result in the shortest possible average response time?

- FIFO minimizes the average response time if the jobs arrive in the ready queue in order of increasing job lengths.
- This avoids short jobs arriving later from waiting substantially for an earlier longer job.
- A special case also exists: when all jobs have the same completion time (job length).

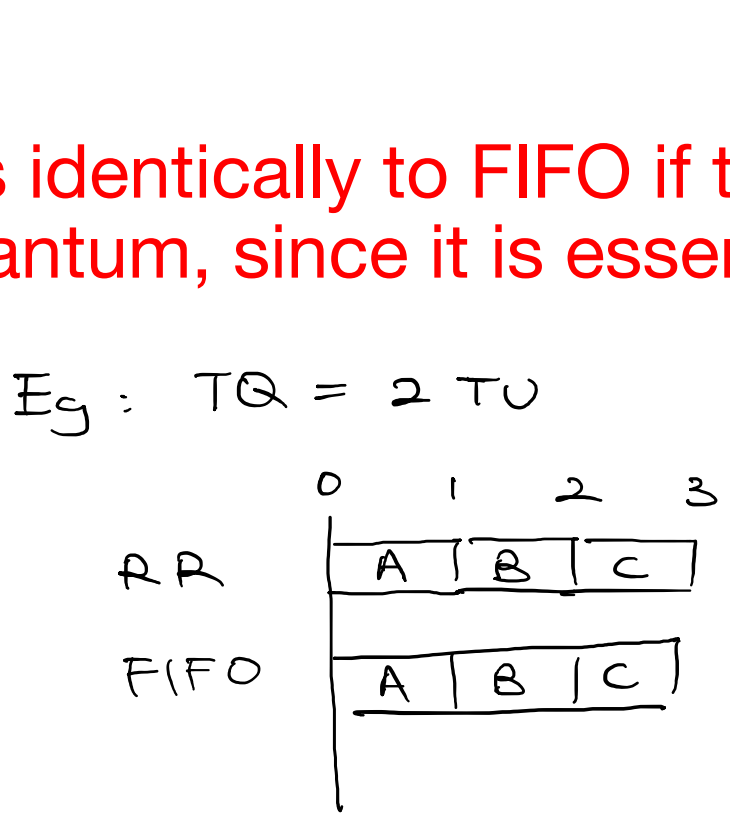
Question 1(b)

Under what conditions does round-robin (RR) scheduling behave identically to FIFO?

Question 1(b)

Under what conditions does round-robin (RR) scheduling behave identically to FIFO?

RR behaves identically to FIFO if the job lengths are shorter than the time quantum, since it is essentially a pre-emptive variant of FIFO.



Question 1(c)

Under what conditions does RR scheduling perform poorly compared to FIFO?

Question 1(c)

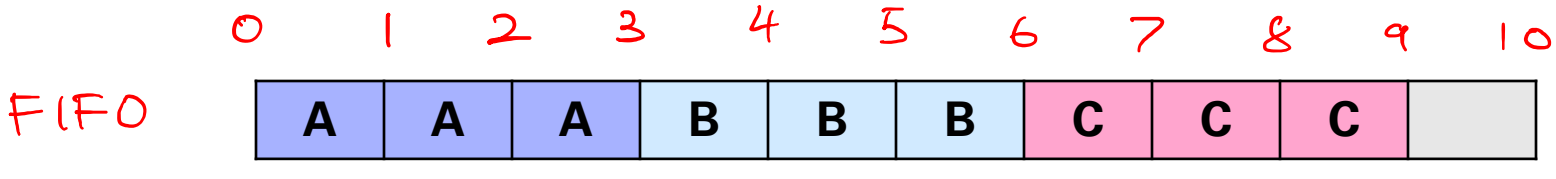
Under what conditions does RR scheduling perform poorly compared to FIFO?

- When the job lengths are all the same and much greater than the time quantum, RR performs poorly in average turnaround time
- When there are many jobs and the job lengths exceed the time quantum, RR results in reduced throughput due to greater overhead from the OS incurred due to context-switches when jobs are pre-empted

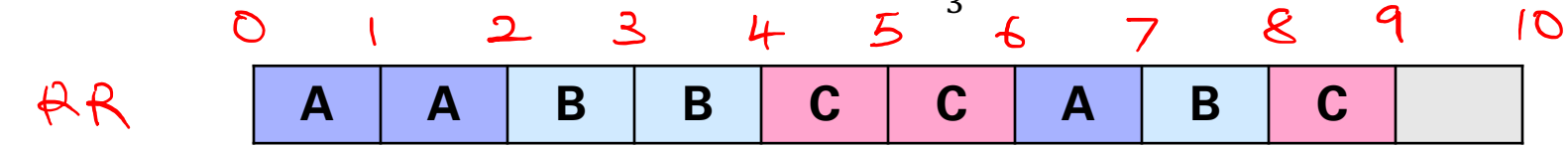
Question 1(c)

Processes A, B and C all have 3 time units of CPU Burst time

Under what conditions does RR scheduling perform poorly compared to FIFO?



$$\text{Avg Turnaround Time}_{FIFO} = \frac{3+6+9}{3} = 6$$



$$\text{Avg Turnaround Time}_{RR} = \frac{7+8+9}{3} = 8$$

Suppose TQ for RR = 2TU

⟶ Processes A, B and C all get pre-empted, resulting in them only being completed later than if there were no interruptions like in FIFO.

Question 1(d)

Does reducing the time quantum for RR scheduling help or hurt its performance relative to FIFO, and why?

- It can both hurt and help performance relative to FIFO.
- Hurt Performance:
  - Reducing the time quantum results in more time spent by the CPU performing context switching.
  - This reduces throughput and increases average turnaround time.
- Helps Performance:
  - Reducing the time quantum reduces the waiting time for a task to first receive CPU time, thus increasing initial response time.

Completion – Arrival time

↳ No. of processes complete execution per unit time

Question 1(e)

Do you think a CPU-bound (CPU-intensive) process should be given a higher priority for I/O than an I/O-bound process? Why?

- Yes: CPU processes can be given higher priority for I/O so they may return to waiting for the CPU, decreasing overall turnaround time at the expense of response time of I/O-bound processes
- No: To maximise responsiveness of I/O-bound processes

Question 2

Question 2: Shared Memory

- In this question, we are going to analyze the pitfalls of having multiple processes accessing and modifying data at the same time.
- Compile and run the code in **`shm.c`**. The executable accepts two command line arguments, n and nChild; if no command line arguments are given, the default values will be used: n is initialized to 100 and nChild is initialized to 1.

Shared Memory: Create

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Creates a shared memory segment of **size** bytes and returns a shared memory identifier.

- **`key_t key`**: A unique key to identify the shared memory segment.
- **`size_t size`**: The size of the shared memory segment in bytes.
- **`int shmflg`**: Flags to control the creation and access behaviour

https://man7.org/linux/man-pages/man2/shmget.2.html

Shared Memory: Attach

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Attaches the shared memory segment associated with the shared memory identifier, **`shmid`**, to the address space of the calling process. It returns the **starting address** of the shared memory segment.

- **`int shmid`**: Shared memory identifier
- **`void* shmaddr`**: To specify a specific memory address to attach the segment to. If left as NULL, it will attach to the first available address as selected by the system
- **`int shmflg`**: Flags to control the creation and access behaviour

https://man7.org/linux/man-pages/man3/shmat.3p.html

Shared Memory: Detach and Destroy

```
#include <sys/shm.h>
int shmdt(const void *shmaddr)
int shmctl(int shmid, int op, struct shmid_ds *buf);
```

The **`shmdt()`** function detaches the shared memory segment located at the address specified by the argument **`shmaddr`** from the calling process's address space.

- **`void* shmaddr`**: Memory address of the shared memory segment.

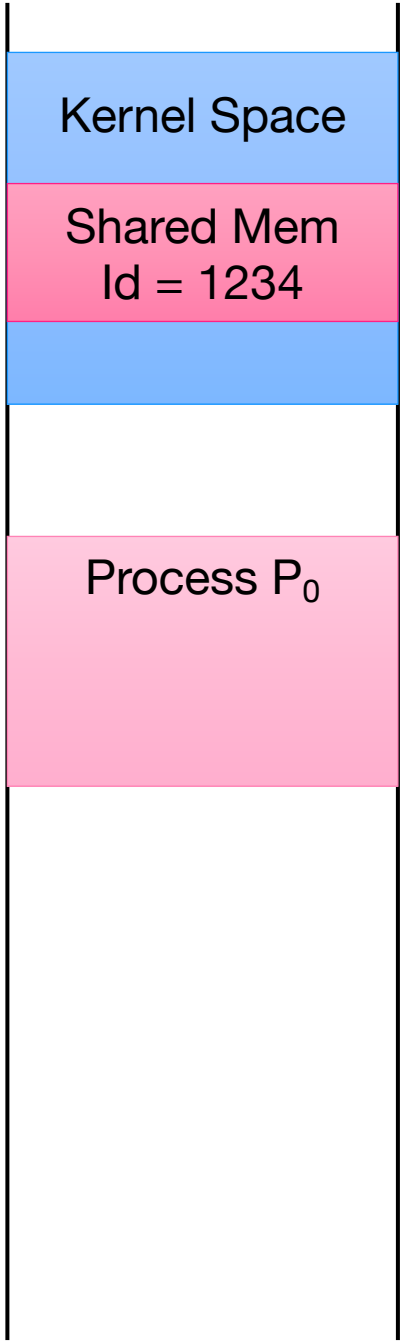
The **`shmctl()`** function performs the control operation specified by op on the shared memory segment whose identifier is given in **`shmid`**.

- When op is set to **`IPC_RMID`**, it marks the shared memory segment to be destroyed after the last process detaches it.

References:
https://man7.org/linux/man-pages/man3/shmdt.3p.html
https://man7.org/linux/man-pages/man2/shmctl.2.html

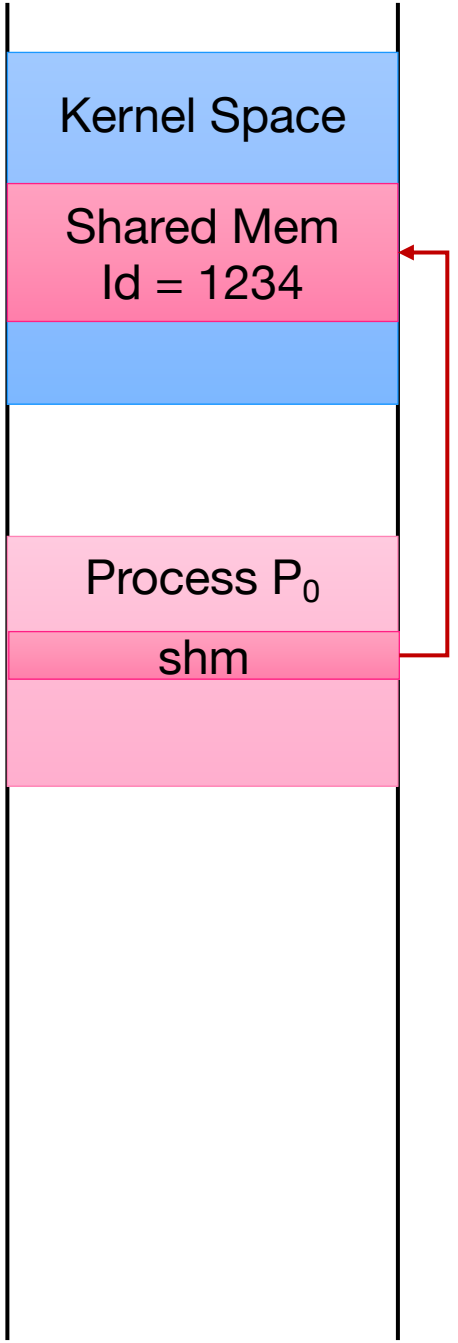
Code Walkthrough

```c
shmid = shmget(IPC_PRIVATE, 1*sizeof(int), IPC_CREAT | 0600);
if (shmid == -1) {
    printf("Cannot create shared memory!\n");
    exit(1);
}
else {
    printf("Shared Memory Id = %d\n", shmid);
}
```



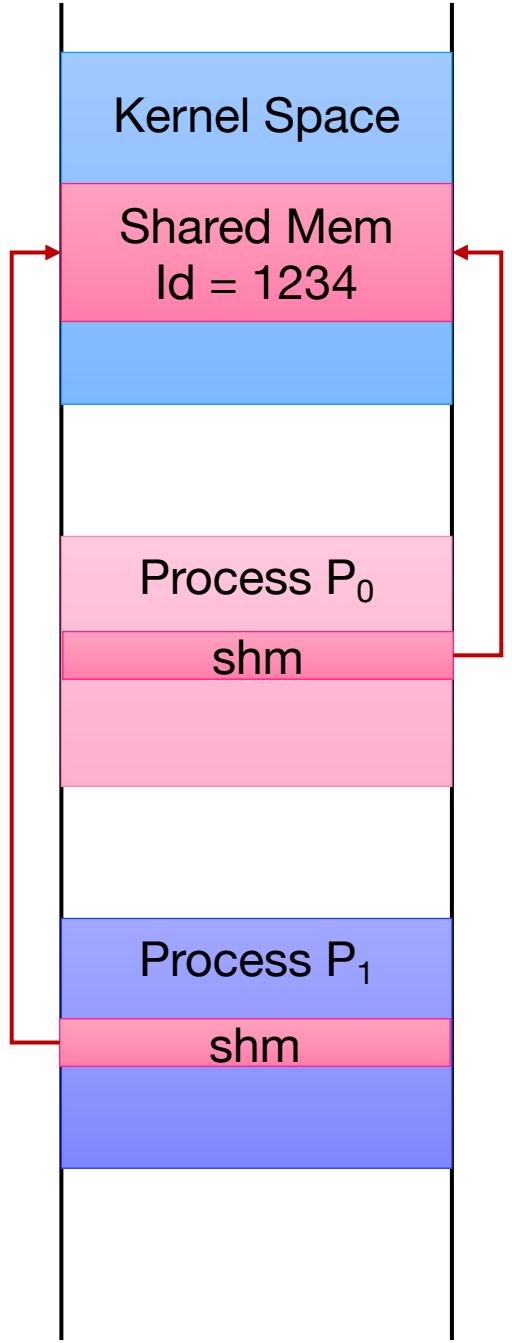
Code Walkthrough

```c
// Attach the shared memory region to this process
shm = (int*)shmat(shmid, NULL, 0);
if (shm == (int*) -1) {
    printf("Cannot attach shared memory!\n");
    exit(1);
}
shm[0] = 0;    // initialize shared memory to 0
```



Code Walkthrough

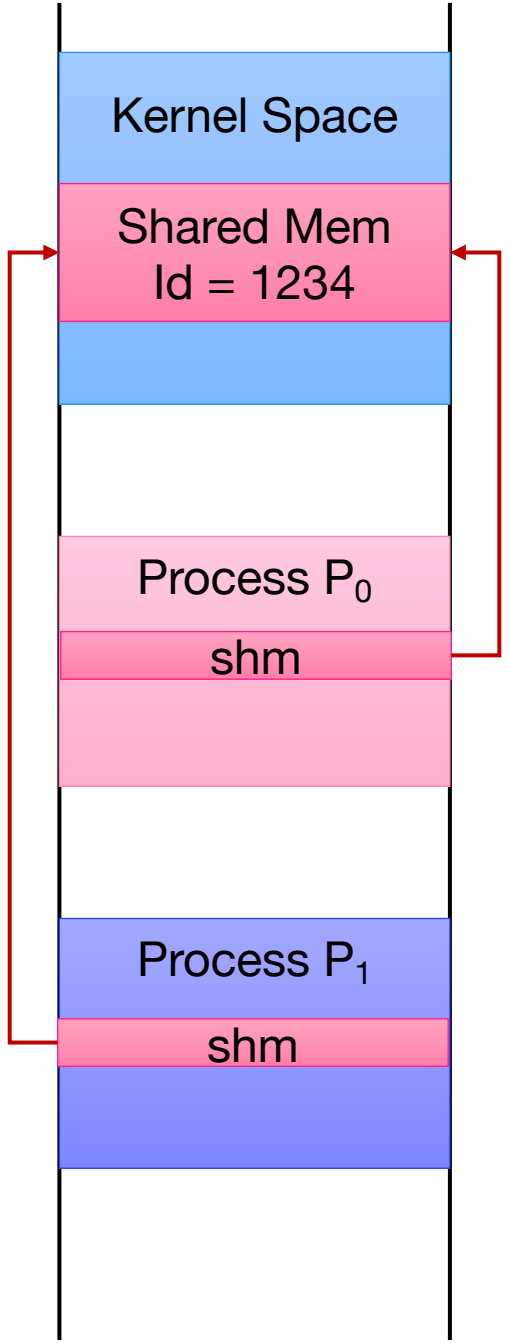
```
for (i = 0; i < nChild; i++) {
    childPid = fork();
    if (childPid == 0)
        break;
}
```



Code Walkthrough

```
for (i = 0; i < n; i++)
    shm[0]++;

if (childPid != 0) {
    for (i = 0; i < nChild; i++)
        wait(NULL);
    printf("The value in the shared memory is: %d\n", shm[0]);
    // detach and destroy
    shmdt((char*)shm);
    shmctl( shmid, IPC_RMID, 0);
}
```



Question 2(a)

The value of the shared memory is initialized to 0, what is the expected final printed value given **n** and **nChild**? (Assuming no interleaving occurs)

Question 2(a)

The value of the shared memory is initialized to 0, what is the expected final printed value given **n** and **nChild**? (Assuming no interleaving occurs)

Expected Final Value: (1 + nChild) * n

Question 2(b)

Run the program multiple times with various values for **n** and **nChild**. Observe the result when n is high (e.g. **n = 10000** and **nChild = 10**). Explain the results.

Question 2(b)

Run the program multiple times with various values for **n** and **nChild**. Observe the result when n is high (e.g. **n = 10000** and **nChild = 10**). Explain the results.

- We would observe the value to be smaller than expected in some test cases when n is sufficiently high.
- For a process to increment the variable, it must
  1. Read the value from memory **`(lw $r1, shm[0])`**
  2. Modify it **`(add $r1, $r1, 1)`**
  3. Write the value back to memory **`(sw $r1, shm[0])`**

Question 2(b)

Race Condition: Unexpected results that occur because the order of events are not enforced.

Run the program multiple times with various values for **n** and **nChild**. Observe the result when n is high (e.g. **n = 10000** and **nChild = 10**). Explain the results.

- If two or more processes have their executions interleaved, it is very likely that one process will overwrite the value written by another process.
- In this case, it is impossible to have a deterministic value.

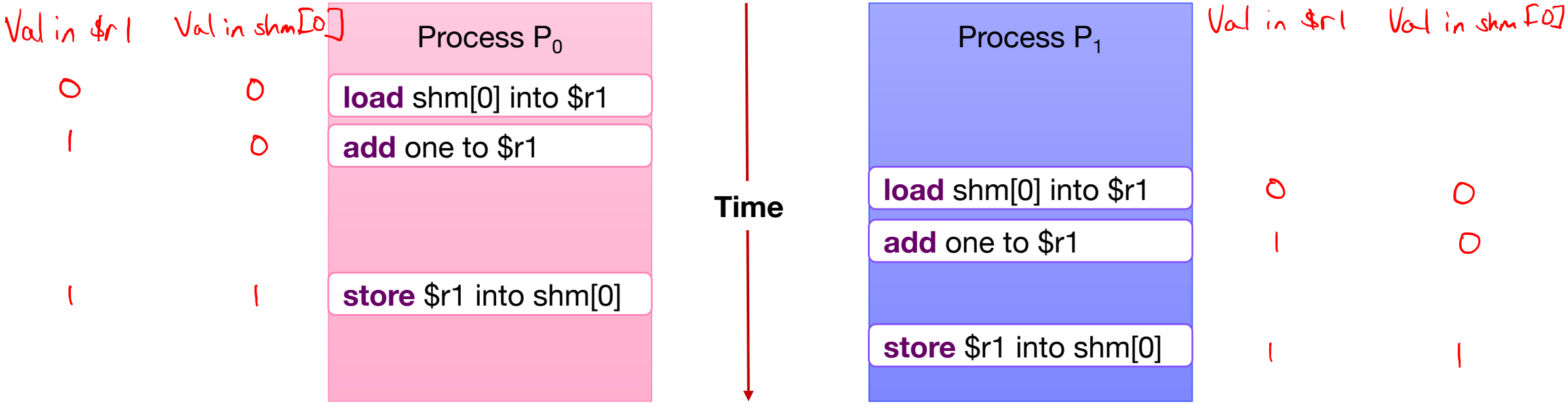
Question 2(b)

Race Condition: Unexpected results that occur because the order of events are not enforced.

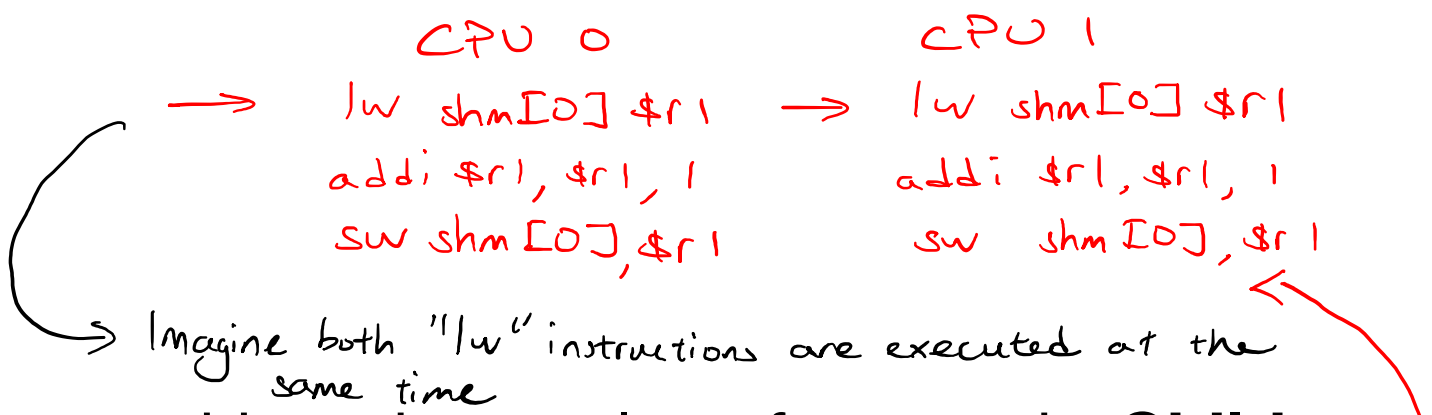
If a process reads from the shared memory before the other process stores the updated value back to the shared memory, then it would have loaded the outdated value into its register $r1

Run the program multiple times with various values for **n** and **nChild**. Observe the result when n is high (e.g. **n = 10000** and **nChild = 10**). Explain the results.

suppose shm[0] is initialized to 0



Question 2(b)



Run the program multiple times with various values for **n** and **nChild**. Observe the result when n is high (e.g. **n = 10000** and **nChild = 10**). Explain the results.

There are two possible scenarios for the interleaving:

1. [Possible on multi-core processors] Multiple processes executes in parallel and happen to interleave the instructions.
2. [Possible on single core processors] Process A swapped out (e.g. due to time quantum) before step (3), the next process B will read the memory value and continue to update. When A swapped back in, its previously read value (stored in register) will be outdated. Upon writing back to memory, A has wipe out work done by B.

Question 2(c)

Do you think (b) is caused by multi-core processors since processes can run in parallel? Would the issue persist if you use a single core processor?

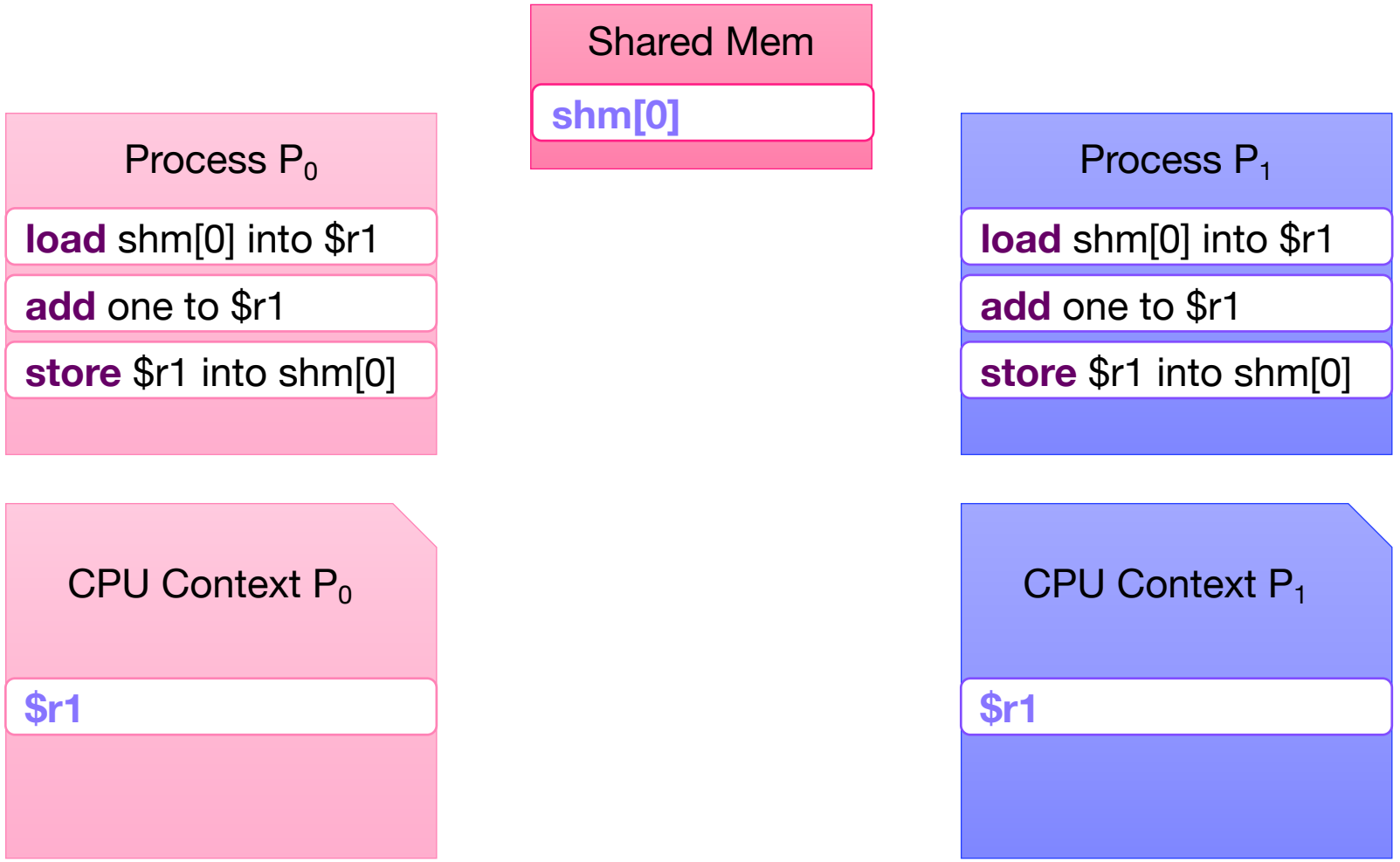
- You can run experiment to find out.
- Use the Linux command taskset to bind a process (and all its child) to a single processor core.
- E.g. "`taskset -c 5 ./a.out 10000 100`" will bind the executable to run on core no. 5.

Question 2(c)

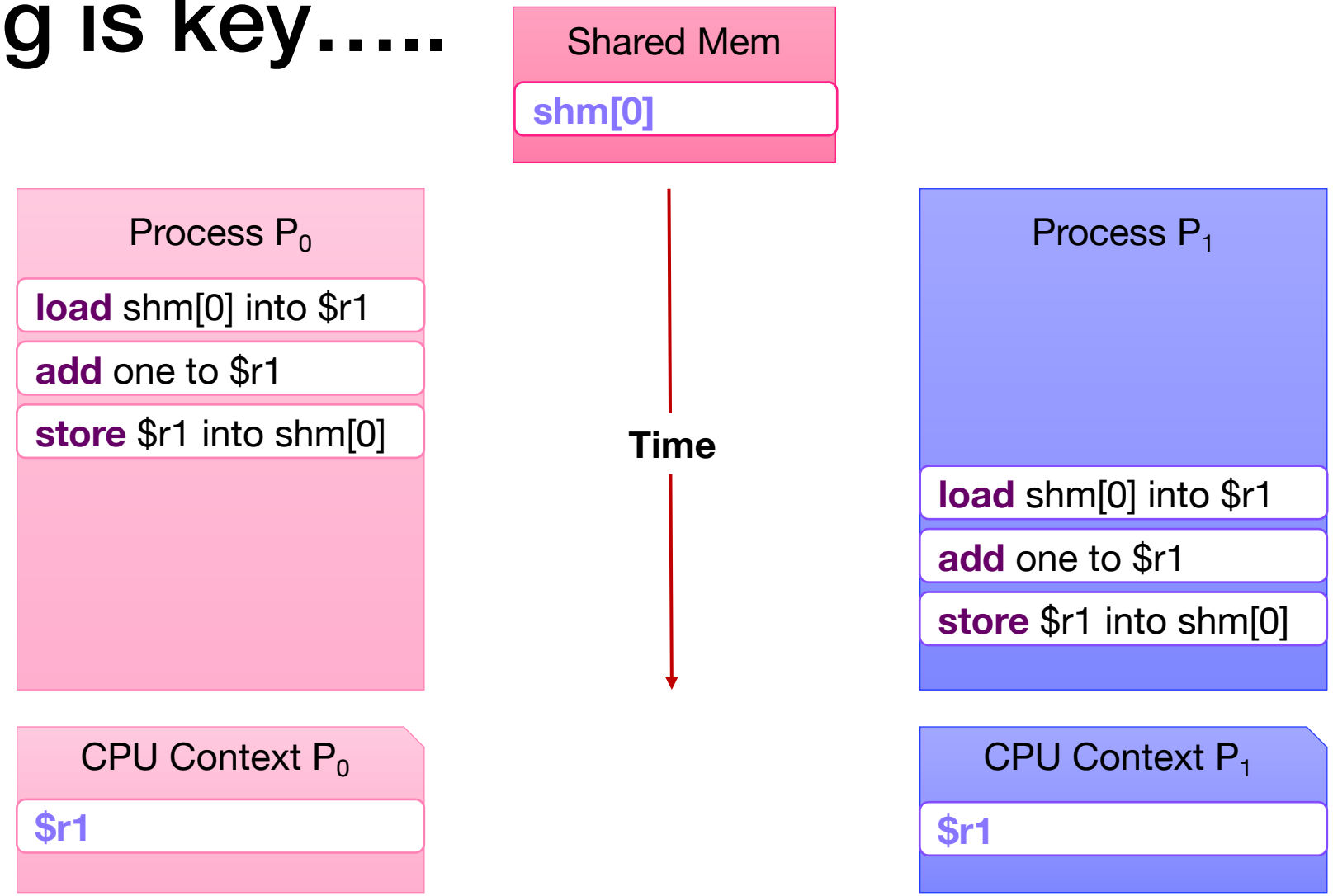
Do you think (b) is caused by multi-core processors since processes can run in parallel? Would the issue persist if you use a single core processor?

You still can get wrong result on a single core. You need to use higher n to increase the chance that a process swapped out at the "right" place for the error to show up.

How is `shm[0]++` performed?



Timing is key…..



Question 3

Question 3: Protecting the Shared Memory

Allowing processes to access and modify shared data whenever they please can be problematic! Therefore, we would like to modify the code in `shm.c` such that the output is deterministic regardless of how large **n** and **nChild** are. More precisely, we want the processes to take turns when modifying the result value that resides in the shared memory.

Process Synchronization

- We need to ensure that only one process can update the shared memory at a time.
- In this week's lecture you will be introduced to process synchronization and mechanisms such as semaphores.
- We can define a critical section to ensure that one process can update shared memory at a time.

You can think of critical sections like going through an automated immigration gate

Question 3: Protecting the Shared Memory

- We will add a new field in shared memory called the **order value**, that specifies which process' turn it is to increment the shared result.
- If the **order value** is **0**, then the parent should increase the shared result, if the **order value** is **1**, then the first child should increase it, if the **order value** is **2**, then the second child and so on.
- Each process has an associated **`pOrder`** and checks whether the value order is equal to its **`pOrder`**; if it is, then it proceeds to increment the shared result. Otherwise, it waits until the order value is equal to its **`pOrder`**.

Question 3: Protecting the Shared Memory

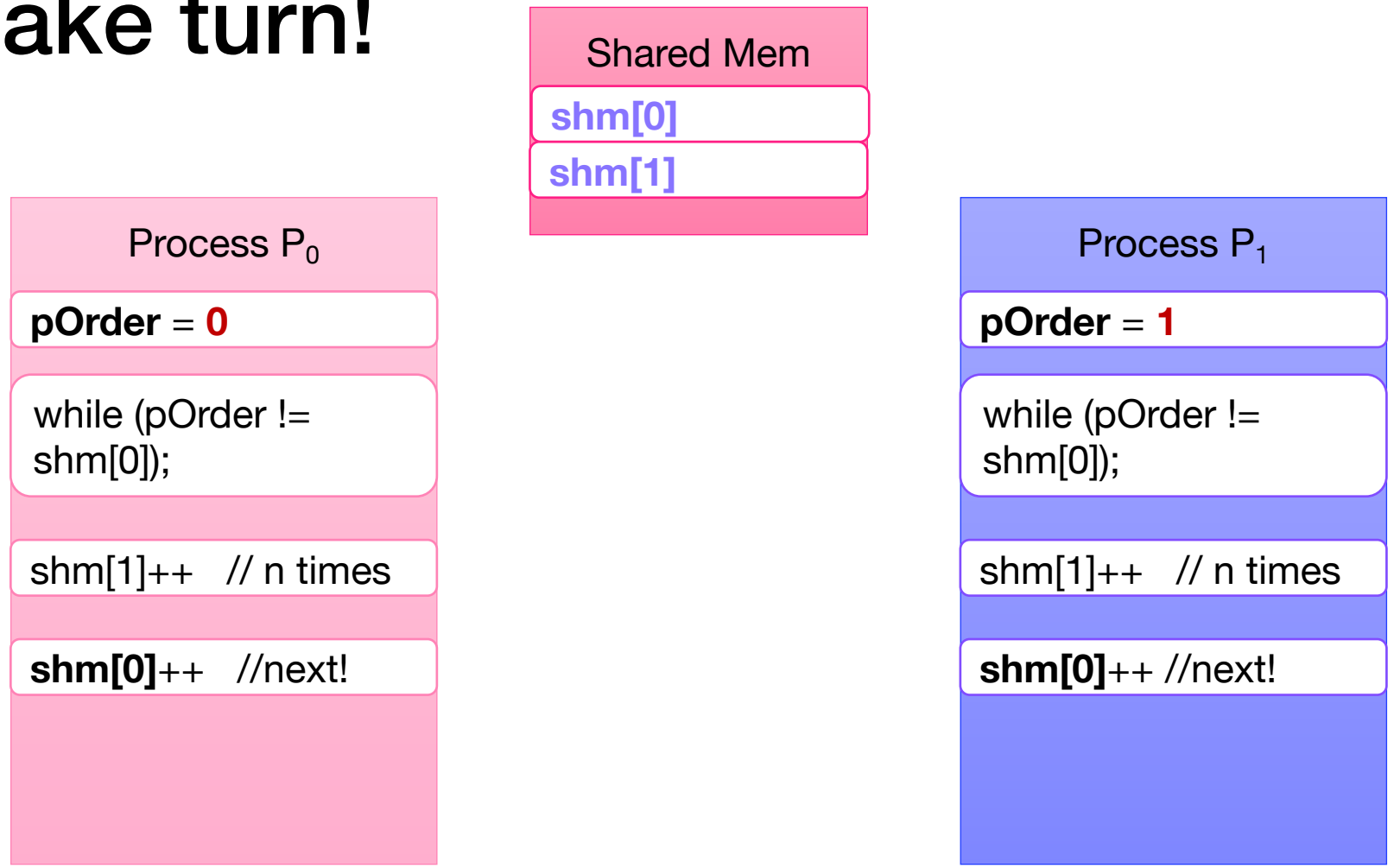
Our Task: Modify the code in **`shm_protected.c`** to achieve the desired result.

i. Create a shared memory region with two locations, one for the shared result, and the other one for the order value;

ii. Write the logic that allows a process to modify the shared result only if the order value is equal to its pOrder (the skeleton already takes care of assigning the right pOrder to each process);

iii. Print the result value and cleanup the shared memory.

Hey, take turn!



Code

```
    // TODO: create a shared memory region that contains two values
    shmid = shmget(IPC_PRIVATE, 2*sizeof(int), IPC_CREAT | 0600);
    if (shmid == -1) {
        printf("Cannot create shared memory!\n");
        exit(1);
    } else {
        printf("Shared Memory Id = %d\n", shmid);
    }

    // TODO: attach the shared memory region to this process
    shm = (int*)shmat(shmid, NULL, 0);
    if (shm == (int*) -1) {
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
```

Code

```
    // TODO: initialize the shared memory
    shm[0] = 0;
    shm[1] = 0;

    for (i = 0; i < nChild; i++) {
        childPid = fork();
        if (childPid == 0) {
          pOrder = i + 1; // each process gets its pOrder
          break;
        }
    }
```

Code

```
    // TODO: only increment the shared value if it's the process' turn
    // don't forget to let the other process know its turn has come
    while (shm[0] != pOrder); // Busy Wait

    for (i = 0; i < n; i++)
        shm[1]++;
    shm[0] = pOrder + 1;

    if (childPid != 0) {
        for (i = 0; i < nChild; i++)
            wait(NULL);
        // TODO: print the result value
        printf("The value in the shared memory is: %d\n", shm[1]);
        // TODO: detach and destroy
        shmdt((char*)shm);
        shmctl( shmid, IPC_RMID, 0);
    }

    return 0;
```

Question 3: Protecting the Shared Memory

Why is shm faster than shm_protected? Why is running shm_protected with large values for nChild particularly slow?

- We made it impossible for two processes to modify the shared memory location at the same time. We did it by serializing the code which will make the execution time longer.
- The technique we use to prevent simultaneous access is called busy waiting. Each process is continuously checking whether its time to modify the variable has come. This requires the process to run on the CPU and consume CPU cycles (thus the name).

Question 3: Protecting the Shared Memory

Why is shm faster than shm_protected? Why is running shm_protected with large values for nChild particularly slow?

Note that at the moment when it's Child X's time to increment the value, there are nChild – X processes doing busy waiting. Each of these processes will get scheduled to run on the CPU but the execution of the program will not make any progress. Because of this, your program will take a long time to complete for large values of nChild.

END OF TUTORIAL