# CS2106
# Introduction to OS

Office Hours, Reading Week

# Topics for Final Exam

**For the final exam, what is the scope?**

- Topics include **everything** we will have covered in the course! However, the accent is going to be on the topics **not covered by the midterm:**
    1. Concurrency Problems
    2. Memory Management
    3. File Systems

- The breakdown between the new content and the old content (midterm-covered) will be 75% : 25%, approximately .

old

## Final Exam Setup: Same as for the Midterm

- Questions will be of various formats. MCQ, true/false, MRQ, fill in, essay (text response).
- The exam will consist of **two LumiNUS quizzes** with a break in between. Your browser must be in the full screen mode for the entire time, **except when you are asking a question**. No electronic materials will be allowed.
- The total time will likely be **120 minutes** (**60min quiz1** + **10min break** for tea/coffee/restroom + **60min quiz2**). There won't be time pressure.
- You can have **10 sheets of paper** (double sided, 20 pages in total) of any content, printed or written. This **includes any scratch paper**.
- SOP: https://docs.google.com/document/d/1s5gD045clpxJwMNUYvJCrWPoNP1z_Lxq214astdiXsY
- If you have any special requirements, please e-mail the profs immediately.

## Examinable or Not?

**Is L12 Slide18 - 20 examinable? Single indirect, double indirect, triple indirect of I-Nodes.**

- Yes.
- You don’t have to memorize the exact details.
  - The necessary details would be provided in the exam

## Interrupts

**For the keyboard interrupt in L13, how does the interrupt handler know which key was pressed? Is it embedded within the interrupt?**

- What's "embedded" in the reception of the interrupt itself is the ID of the device that triggered the interrupt
- This results in the execution of the correct interrupt handler
- However, the information about the key that's pressed is usually in some device buffer
- The interrupt handler is then going to read that buffer and notify the relevant application about it.

## Interrupts

**Are interrupts caught by the CPU or by the operating system?**

- CPU is the first to be notified about the interrupt
- The execution is quickly transferred to the right handler
  - With some hardware help (mainly looking up the interrupt vector table)
- The interrupt handlers are considered to be part of the OS and are executed in kernel mode.

## Interrupts

div 0

TRAP

**Can I confirm that exceptions and software interrupts (syscalls) are generated by the CPU?**

- They are the result of executing an instruction of the program.
- That execution happens in the CPU, where the event is generated
  - But the actual trigger is, for example, division by zero or the instruction being a syscall trap instruction

**Are these exceptions and syscalls caught by the operating system?**

- The OS is not “catching” them or “detecting” them.
- OS is notified about them and executes the predefined routines in response.

## Interrupts

**During an interrupt, does the hardware or handler save registers into the stack?**

- Usually, the hardware saves only the PC and the status register
- The rest is saved by the routine

**This stack refers to the stack of the process that was running?**

- Yes

**And who restores these registers once the handler is done?**

- The routine restores the registers it saved,
- the rest is restored by the hardware

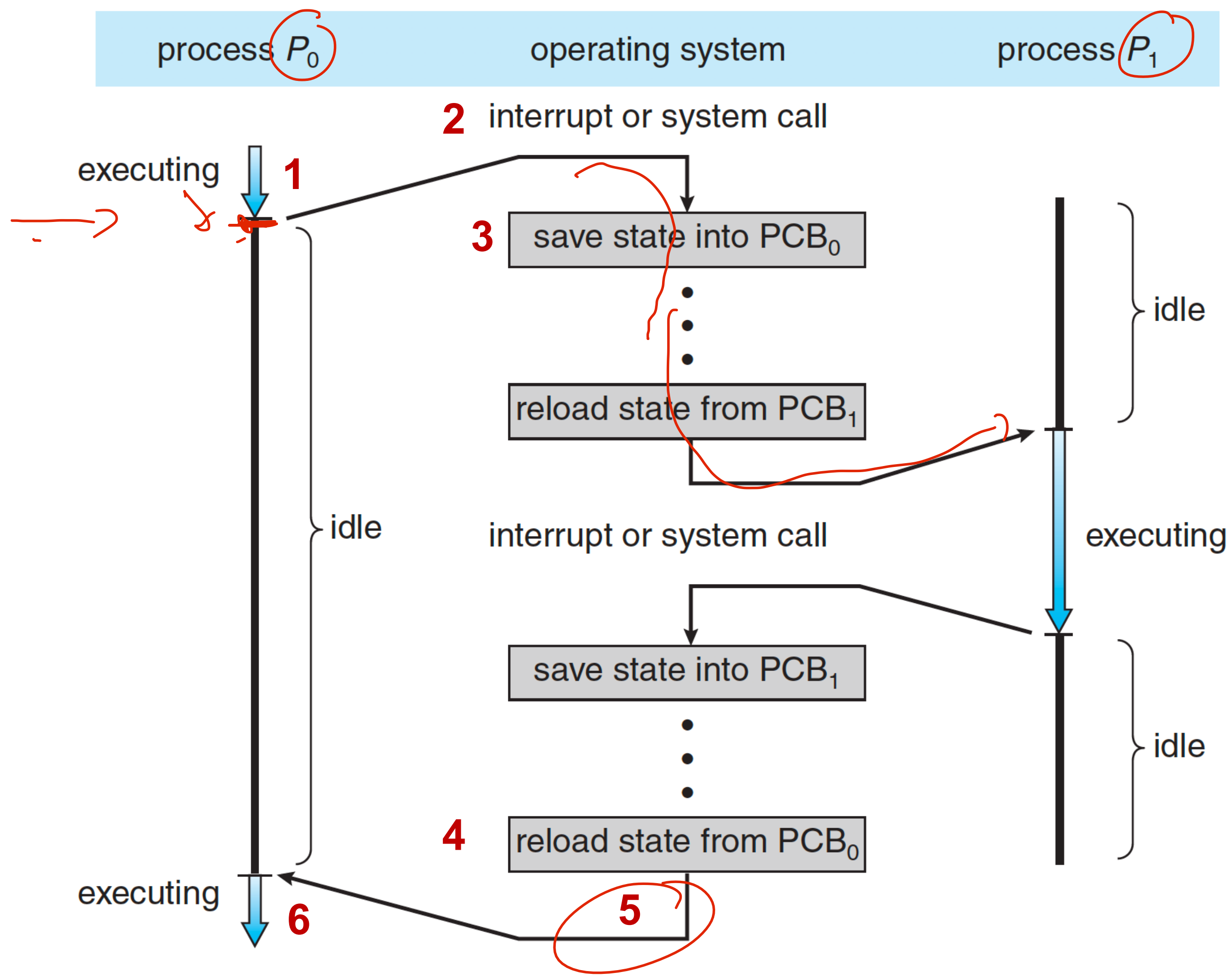
## Mock Midterm PCB

**Prof can you go through the mock midterm quiz question on PCB again?**

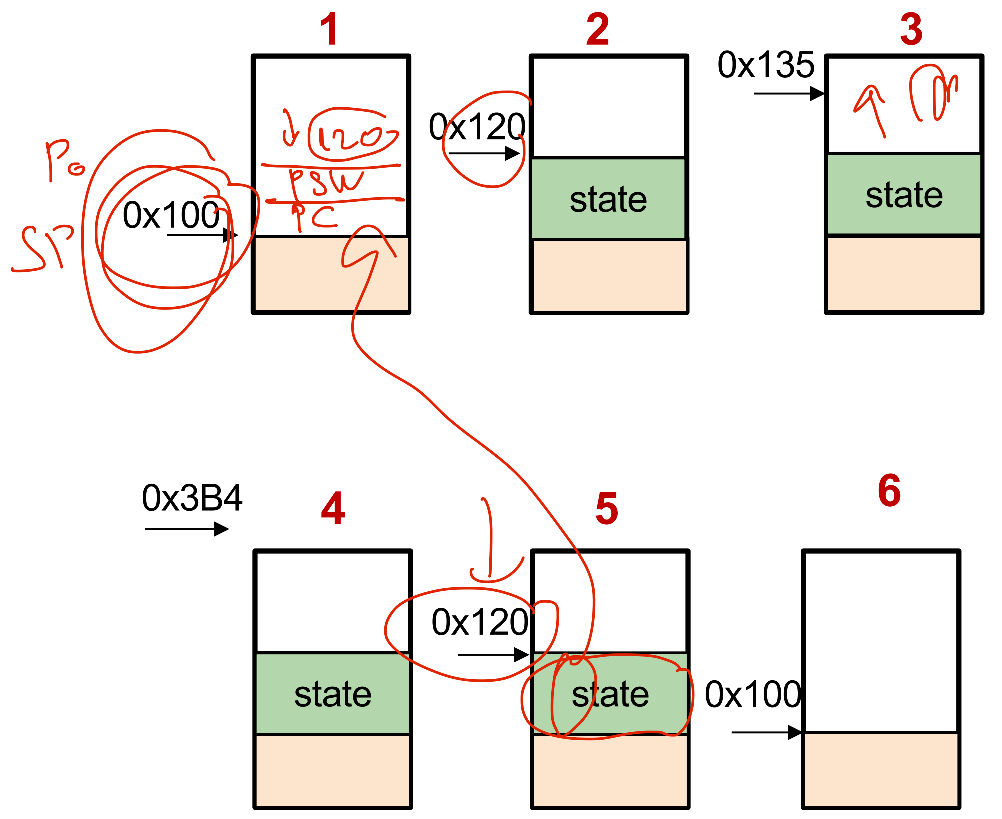
- Please feel free to ignore it
- The question was too tricky and nitty gritty
  - This was intentional, because it was for bonus
- Not really necessary for you to understand for the exam

P0
PCB: $\boxed{SP = ?}$

~~100~~
120



Stack of Process P0 and value of stack pointer

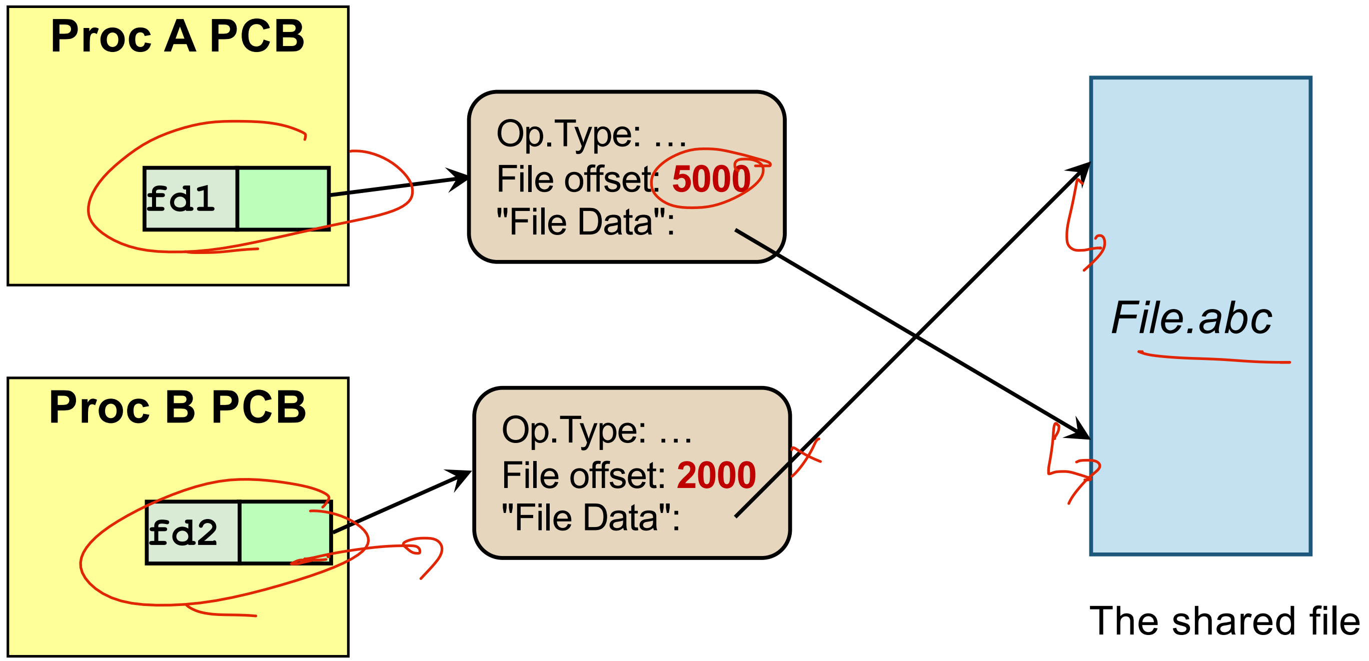


Check the notes for explanation

## “ls”

**When we press “ls”, the shell will show the characters "ls", doesn't this require that the shell transitions into the running state and gets blocked again?**

- Yes,
- The shell process will get into the running state for a little bit of time after every key pressed
  - To print the character
  - Note that the the printing of characters will also block it the shell



The shared file

# Processes Sharing an Open File in Unix: Case 1

**Isn't the system wide table supposed to contain only 1 entry per unique file?**

- In Linux there could be multiple openings of the same file by different processes

**Different file offset values means unique?**

- Each opening has a file offset that can independently advance

**Will this file offset value be constantly changing when it's reading the file?**

- It will change only for that particular entry (i.e., opening) of the file
- Not for other openings of the same file

## Research with DNA

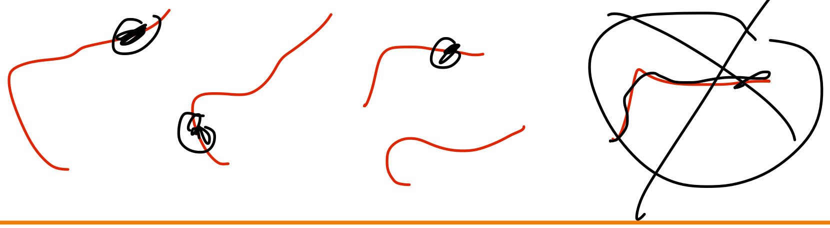
**if I’m interested in the research section of "programming in DNA", is there like a focus section in SoC or any jobs I should look out for or any profs doing research in this field? :)**

- Currently only Prof. Diptarka Chakraborty and myself do research related to DNA data storage at SoC
- You can contact us if interested for collaboration

## Loss of DNA molecules

**DNA reading seems quite stochastic. What if a specific data chunk is not read after the many tries? Or is the chance of not reading the required chunk very small?**

- Great questions
- The situation is called “data erasure”, and happens often
- Complex error-correcting codes are used in a complicated manner to recover from such errors
- These codes must be stripped across many DNA molecules so that they allow for some of these molecules to be lost



# Sleeping Barber

We explore ***the sleeping barber*** problem.

- The sleeping barber has a barbershop with **10 chairs** for customers who wait for a haircut, and the barber chair.
- If a customer enters, and all the waiting chairs are occupied, then the customer queues outside the shop. If the barber is busy, but a waiting chair is available, then the customer sits in one of the free chairs.
- When there are no customers, the barber sleeps. Customers sit in the barber chair during the haircut, allowing more customers to wait seated.

## Sleeping Barber

Semaphores

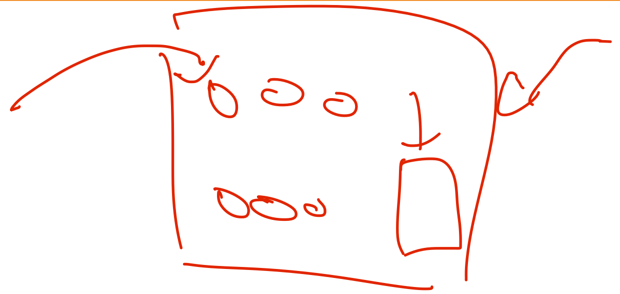
**a) Complete the code skeleton below to synchronize the barber and customer processes.**

```
void barber(){
    while(1)
     {
          ...
        cutHair();
          ...
      }
}
```

```
void customer() {
   while(1)
    {
     ...  //visit the barber shop

     delay(rand()); // wait for a random amount of
                    // time between 2 shop visits
    }
}
```

# Sleeping Barber



10

```
#define    CHAIRS   10
Semaphore waitingChairs = CHAIRS;
Semaphore barberReady   = 1;
Semaphore customerReady = 0;
Semaphore haircutDone   = 0;
```

→ Safe distancing

# Sleeping Barber

```
void barber(){
    while(1)
     {
         P(customerReady);
         cutHair();
         V(haircutDone);
         V(barberReady);

     }
}
```

```
#define    CHAIRS   10
Semaphore waitingChairs = CHAIRS;
Semaphore barberReady   = 1;
Semaphore customerReady = 0;
Semaphore haircutDone   = 0;
```

## Sleeping Barber

```
#define    CHAIRS   10
Semaphore waitingChairs = CHAIRS;
Semaphore barberReady   = 1;
Semaphore customerReady = 0;
Semaphore haircutDone   = 0;
```

safe distancing

```
void customer(){
  while(1)
  {
     P(waitingChairs);
     P(barberReady);
     V(waitingChairs);
     V(customerReady);
     P(hairCutDone); // wait for the haircut to finish
     delay(rand());  // wait for a random amount of
                       time between barbershop visits
  }
}
```

## Sleeping Barber – Leave instead of queueing

b) Modify your code to allow for customers to leave the shop if there are no free chairs, instead of queueing outside.

```
Semaphore waitingChairMutex  = 1;
Semaphore barberReady        = 1;
Semaphore customerReady      = 0;
Semaphore haircutDone        = 0;

int availableChairs = CHAIRS;
```

# Sleeping Barber

```
void barber(){
    while(1)
     {
        P(customerReady);
        P(waitingChairMutex);
        availableChairs++;
        V(waitingChairMutex);
        cutHair();
        V(haircutDone);
        V(barberReady);
     }
}
```

```
Semaphore waitingChairMutex      = 1;
Semaphore barberReady            = 1;
Semaphore customerReady          = 0;
Semaphore haircutDone            = 0;


int availableChairs = CHAIRS;
```

# Sleeping Barber

```
Semaphore waitingChairMutex      = 1;
Semaphore barberReady            = 1;
Semaphore customerReady          = 0;
Semaphore haircutDone            = 0;


int availableChairs = CHAIRS;
```

```
void customer(){
   while(1)
   {
         P(waitingChairMutex);
         if(availableChairs==0)
              V(waitingChairMutex); // leave the shop
         else{
              availableChairs--;
              V(waitingChairMutex);
              P(barberReady);
              V(customerReady);
              P(hairCutDone); // wait for the haircut to finish
         }
         delay(rand()); // wait for a random amount of time
                        // between barbershop visit
   }
}
```

## Sleeping Barber: Message Passing Solution

c) Synchronize the barber and customer processes using asynchronous message passing. Assume that messages are passed through message queues of finite capacity, specified at the creation time, with sending (receiving) processes blocking when the queue is full (empty).

## Initialization

10

```
#define        CHAIRS  10

MessageQueue chairQueue = createQueue(CHAIRS, sizeof(DummyMessage));

MessageQueue barberQueue = createQueue(1, sizeof(DummyMessage));

MessageQueue customerQueue = createQueue(1, sizeof(DummyMessage));

MessageQueue haircutDoneQueue = createQueue(1, sizeof(DummyMessage));
```

# Barber

```
void barber(){
    DummyMessage m;
    for(int i=0; i< CHAIRS; i++)
            send(chairQueue, &m);
    while(1)
     {
            send(barberQueue, &m);
            receive(customerQueue, &m);
            cutHair(); 2
            send(haircutDoneQueue, &m);


     }
}
```

```
#define    CHAIRS   10
MessageQueue chairQueue; [10]
MessageQueue barberQueue;
MessageQueue customerQueue;
MessageQueue haircutDoneQueue;
```

## Customer

```
#define    CHAIRS   10
MessageQueue chairQueue;
MessageQueue barberQueue;
MessageQueue customerQueue;
MessageQueue haircutDoneQueue;
```

```
void customer(){
   while(1)
   {
       DummyMessage m;
       receive(chairQueue, &m);
       receive(barberQueue, &m);
       send(chairQueue, &m);
       send(customerQueue, &m);
       receive(haircutDoneQueue); // wait for the haircut to finish

       delay(rand());  // wait for a random amount of
                       // time between barbershop visits
   }
}
```

Safe [illegible]

semaphore here

send, MD recv

recv.

## Notes

- Note that this message passing solution is not as elegant as the solution to the producer-consumer problem.

- The producer-consumer problem is essentially a communication problem, where the producers communicate items to consumers;
  - In shared memory, communication happens through a shared buffer, in which case synchronization is required
  - In message passing, it happens through OS-managed message queues, in which case the explicit synchronization not needed.

- Message passing is naturally a  perfect match for producer-consumer.

## Notes

- In contrast, in the sleeping barber problem, no data needs to be communicated at all – it's all about synchronizing the actions of different processes around shared resources (chairs).

- We can see that in the fact that none of the messages exchanged carry any information. That's why the message passing primitives are not a natural fit for this problem and relies on emulation of synchronization primitives such as semaphores.

Thank you!