

CSC369 Week 2 Notes

Hyungmo Gu

May 22, 2020

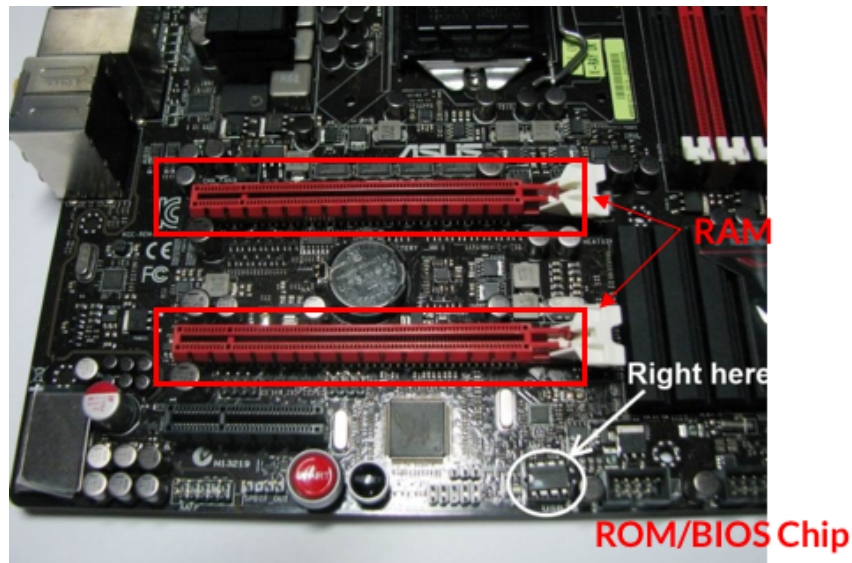
1 System Calls

- Bootstrapping
 - Bootstrapping

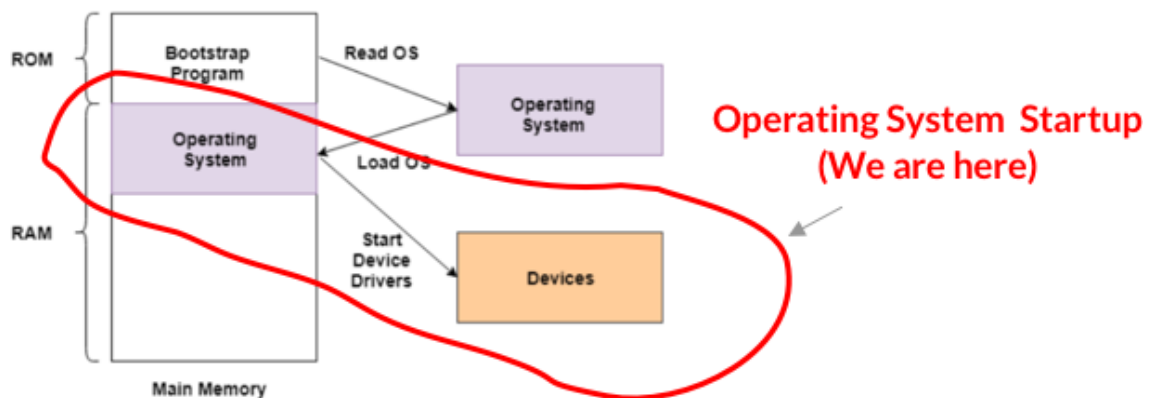


- * executes **Bootstrap Program**
 - is the first code that runs when the computer system is started
- * Entire operating system depends on the bootstrap program to work correctly
- * Locates and loads kernel (code of operating system) onto RAM
 - kernel = code of the operating system
 - kernel is in HDD
- * Bootstrap program is in ROM
- ROM
 - * is called **read-only-memory**
 - * Is also called **BIOS chip** (Basic Input/Output System)

- * is non-volatile
- * is stored in motherboard



- Operating System Startup



- Initializes OS

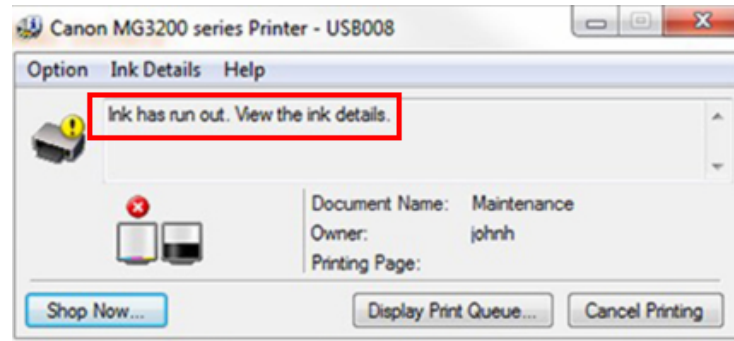
- * Initialize internal data structures
- * Create first process
- * Switch mode to user and start running first process
- * Wait for something to happen

- Requesting OS Services

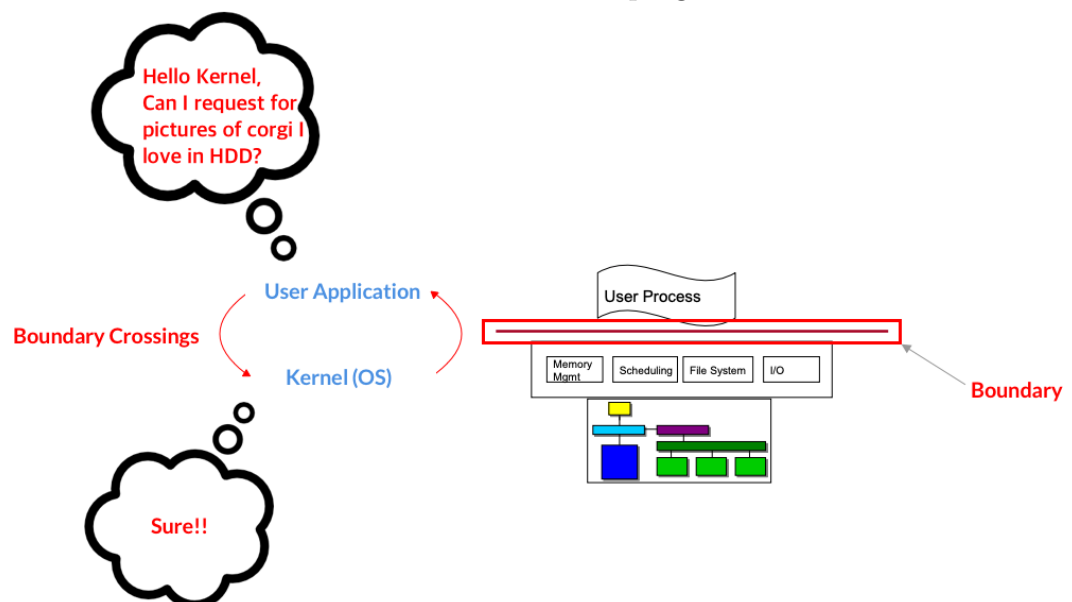
- Some services offered by OS are:

- * Program execution
 - Loading program to memory and executing program

- * I/O operations
 - Keyboard, mouse, speaker
- * File system manipulation
 - Reading and writing files and directories
- * Error Detection
 - Error that pops when printer ink is empty



- Operating system and user programs are isolated
- How do they communicate?
- Boundary Crossings
 - Boundary
 - * Is the line between user applications and kernel
 - * Data is difficult to move back and forth between this line
 - Boundary Crossings
 - * Is the communication that occurs between a program and kernel

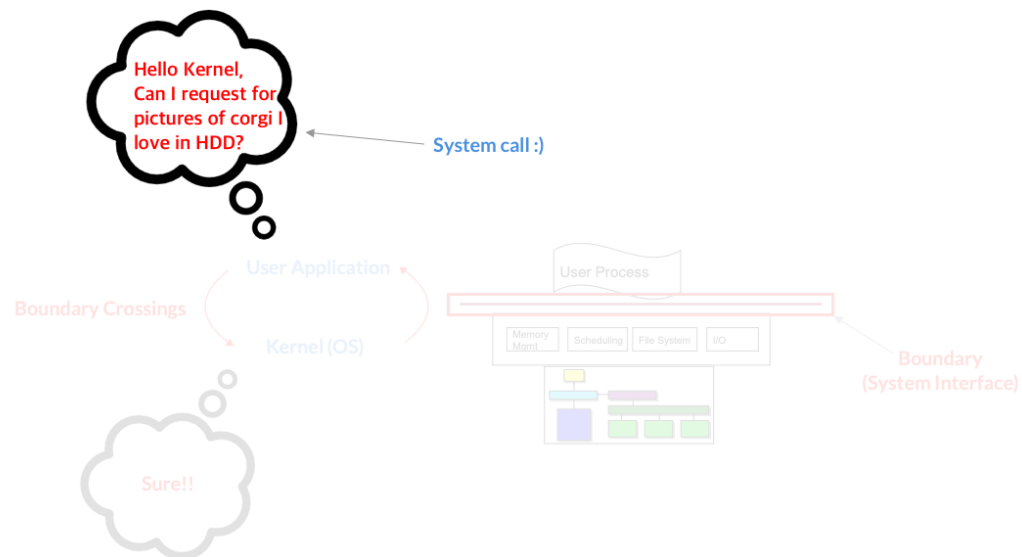


- * Communication occurs by sending data from one program into kernel, and then back
- More can be found here
- System Calls for Process Management
 - Major system calls

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

- Wait, System Calls?

System Calls are interrupt signals sent by software



- * Is a programmatic way of a program requesting for service to kernel of operating system
- * It's like 'Hey OS, could you do *y*? It's really important'
- * i.e. Fetching a file in hard-disk drive

- System Calls for File Management

- Major system calls

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing, or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	get a file's status information

- System Call Interface

- Interface

- * Is a point where two systems, subjects, organizations, etc. meet and interact. (Definition)

- System Call Interface

- * Is the point where user mode and kernel mode meet and interact

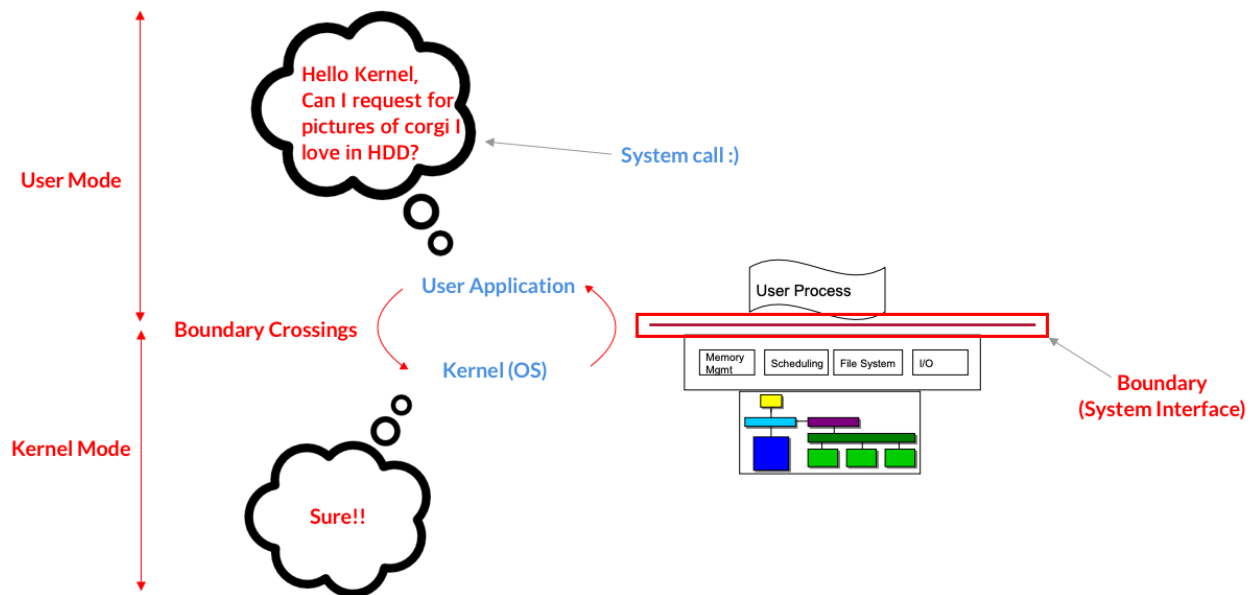


Figure 1: Now, there really is a party going on here :)

- * Most of the system call interface are hidden from the programmer by API

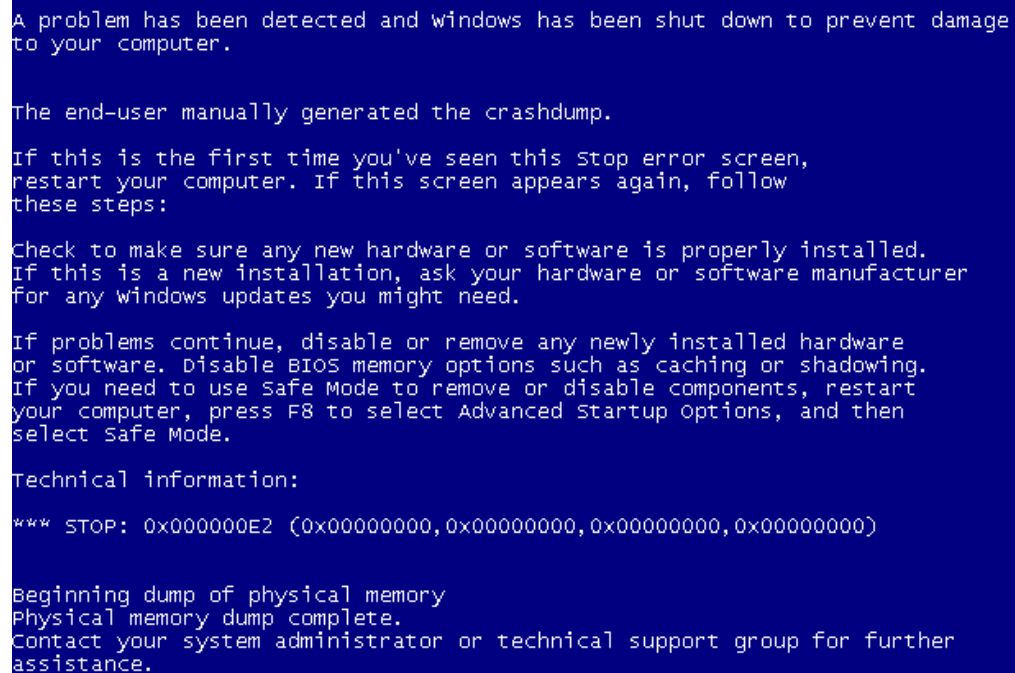
- User Mode

- * Cannot directly access memory and other hardware
 - * Is safe
 - * Crash → doesn't halt entire system

- Kernel Mode

- * Does have access to memory and other hardware
 - * Is a privileged mode

- * Is not safe
- * Is fragile
- * Crash → halts entire system
 - i.e. The Blue Screen of Death >:)



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

The end-user manually generated the crashdump.

If this is the first time you've seen this stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x000000E2 (0x00000000,0x00000000,0x00000000,0x00000000)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.
```

- System Call Operation
 - Register is a local storage device present in CPU
 - Register may include the address of the memory location instead of real data
 - CPU takes data that has to be processed from register

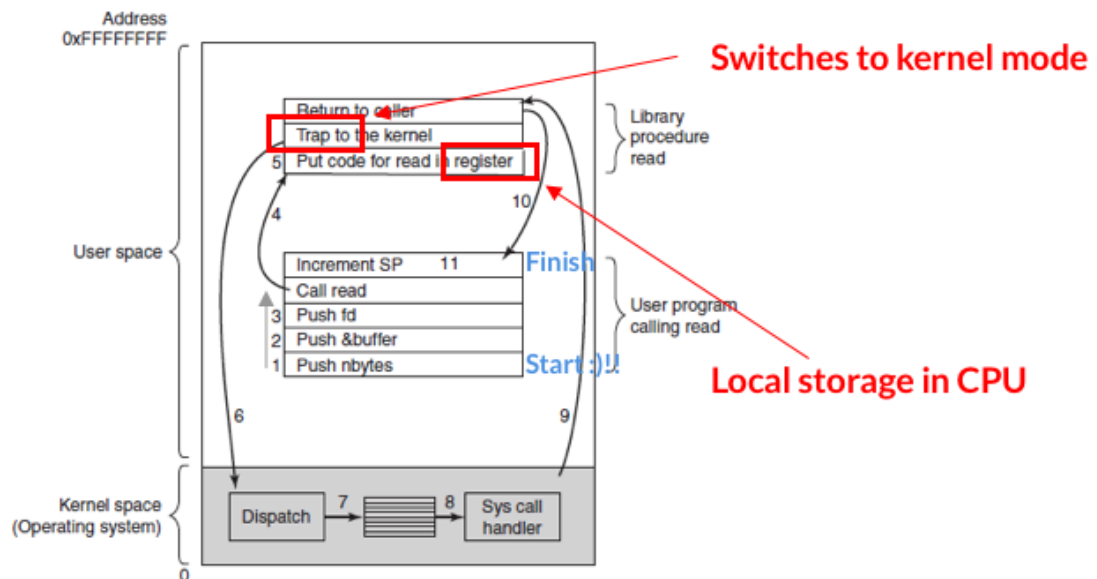
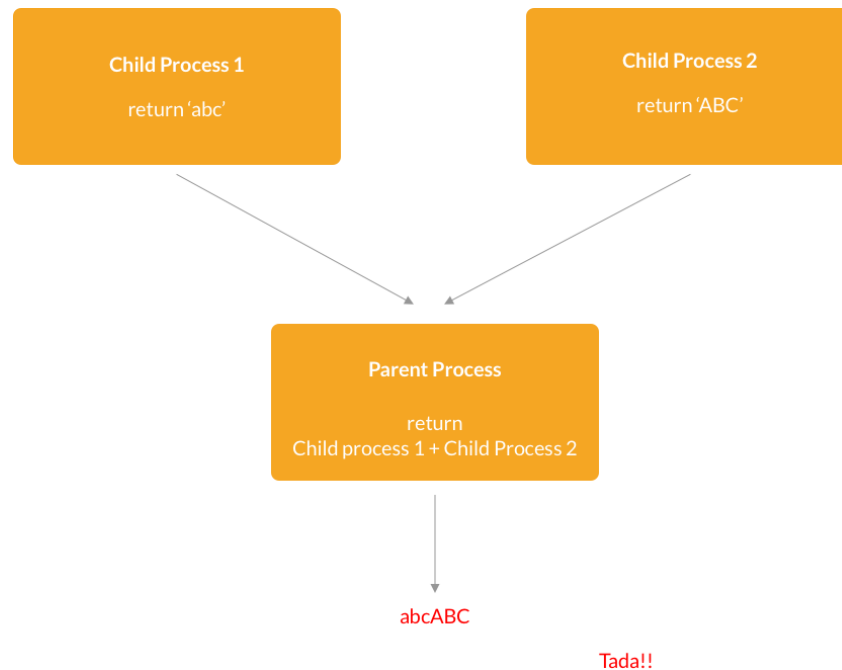


Figure 1-17. The 11 steps in making the system call `read(fd, buffer, nbytes)`.

2 Intro to Synchronization

- Cooperating Process
 - Independent Process
 - * Cannot affect or be affected by other processes
 - * i.e. Running *hello_world.c* program
 - Cooperating Process
 - * is not independent
 - * must be able to communicate and synchronize actions
 - * i.e. One child process outputs 'abc', another child process outputs "CBA", and parent combines together to outputs 'CBAabc' :)



- Interprocess Communication

- Cooperating processes need to exchange information using either

- * Shared memory (e.g. fork())
- * Message sharing

- Message passing models

- * Send(P, msg) - Send msg to process P
- * Receive(Q, msg) - Receive msg from process Q

- Motivating Example (importance of communication in cooperating process)

- ATM machine

- * Suppose the bank has following functions

```

Withdraw(acct, amt) {
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct, balance);
    return balance;
}
  
```

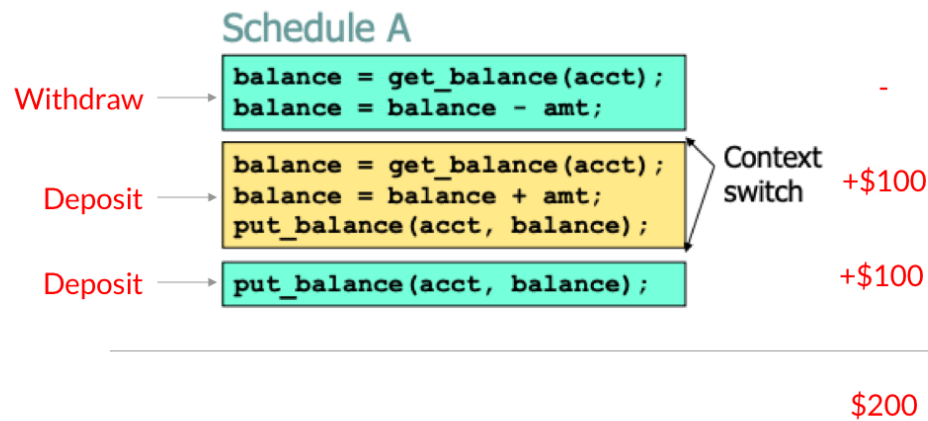
Withdraw

```

Deposit(account, amount) {
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct, balance);
    return balance;
}
  
```

Deposit

- * And suppose Moe deposits \$100 and my love withdraws \$100 on two different ATM machines using the same account.
- * Without communication, processes become interleaved
- * Then,



We are \$100 richer!!



- What Went Wrong
 - Two concurrent threads manipulated a *shared resource* (the account) without synchronization.
 - *Race condition* occurred
 - * Outcome depends on the order in which accesses taking place
 - Fix → Mutual Exclusion
 - * Allow resource be manipulated only one thread at a time



- The Plan - Mutual Exclusion

- Is a program object that prevents simultaneous access to a shared
- Is used to avoid the *race condition* resource
- So, for this problem
 - * Given
 - A set of n threads, T_0, T_1, \dots, T_n
 - A set of resources shared between threads
 - A segment of code which accesses the shared resource, called the *critical section*

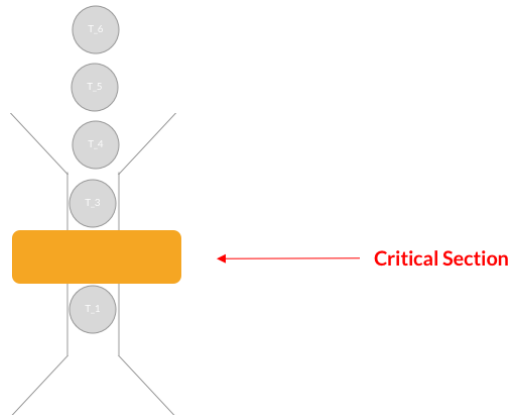
```

Withdraw(acct, amt) {
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct, balance);
    return balance;
}

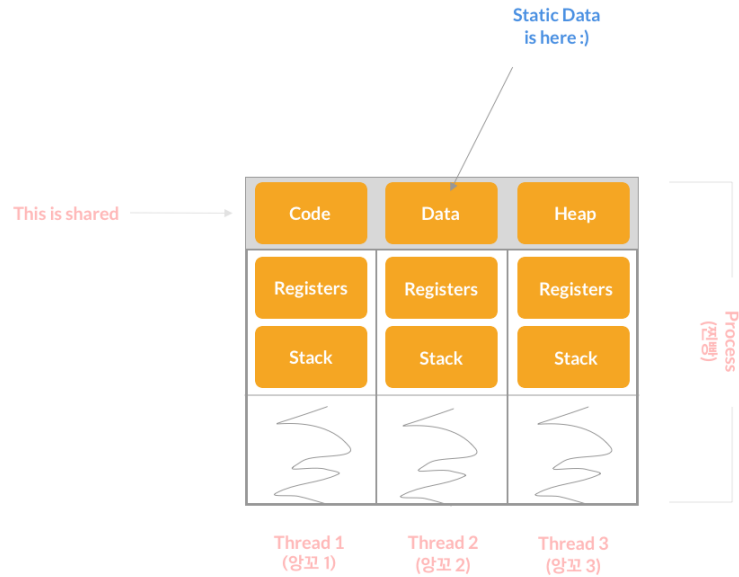
```

← Critical Section

- * We want to ensure that
 - Only one thread at a time can execute in the critical section
 - All other threads are forced to wait on entry
 - When a thread leaves the CS, another can enter

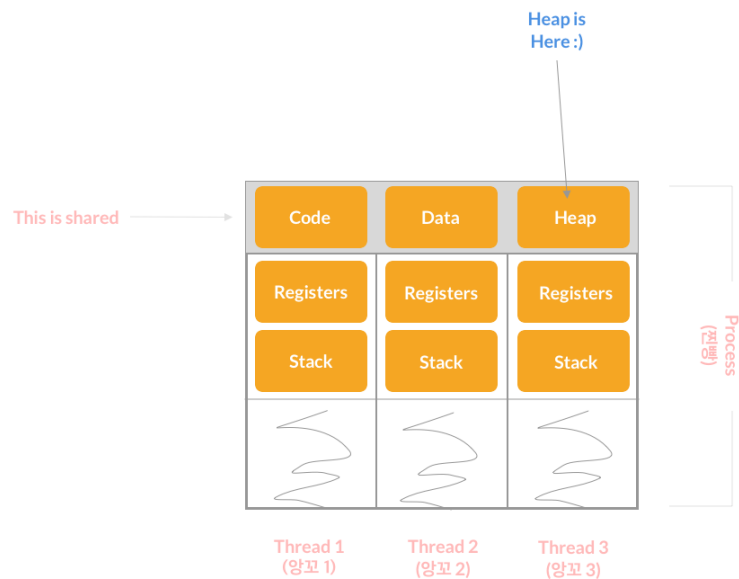


- Aside: What program data is shared between threads?
 - Shared
 1. Local Variables
 - * Are stored in their own private stack
 - Not Shared
 1. Global Variables
 - * Are stored in Static Data segment, accessible by any thread



2. Dynamic Objects and other heap objects

- * Allocated from heap with malloc/free or new/delete



• Critical Section Requirements

1. Mutual Exclusion

- If one thread is in the CS, then no other is

2. Progress

- If no thread is in the CS, and some threads want to enter CS, it should be able to enter in definite time

3. Bounded Waiting (No Starvation)

- Means no process should wait for a resource for infinite amount of time.

4. Performance

- The overhead of entering and exiting the CS is small with respect to the work being done with it

• 2-Thread Solutions: 1st Try

```

My_work(id_t id) { /* id_t can be 0 or 1 */
    ...
    while (turn != id) ; /* entry section */
    /* critical section, access protected resource */
    turn = 1 - id; /* exit section */
    ... /* remainder section */
}

```

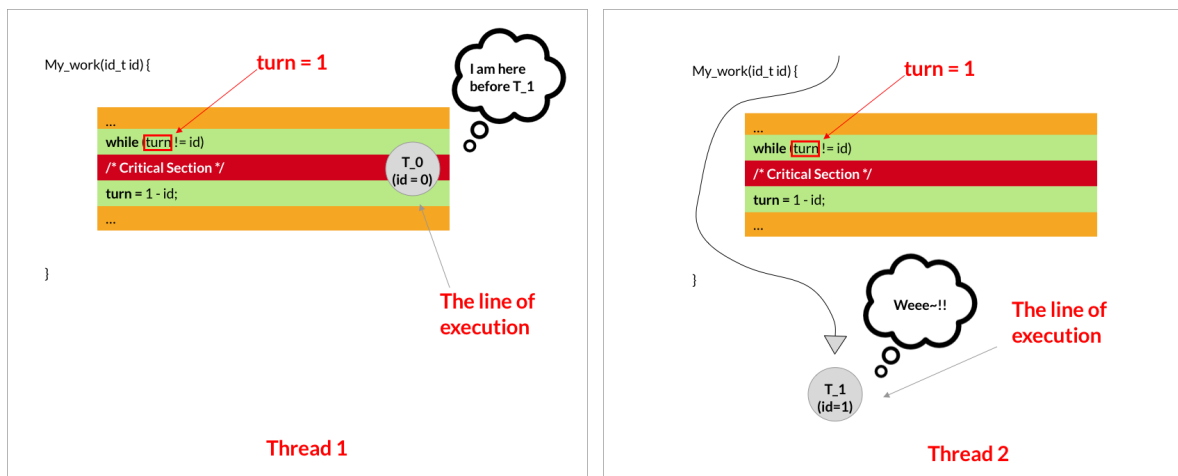
- Let the threads share an integer variable *turn* initialized to 0 (or 1)
- If $turn = i$, thread T_i is allowed into its CS

GOOD Mutual Exclusion is satisfied

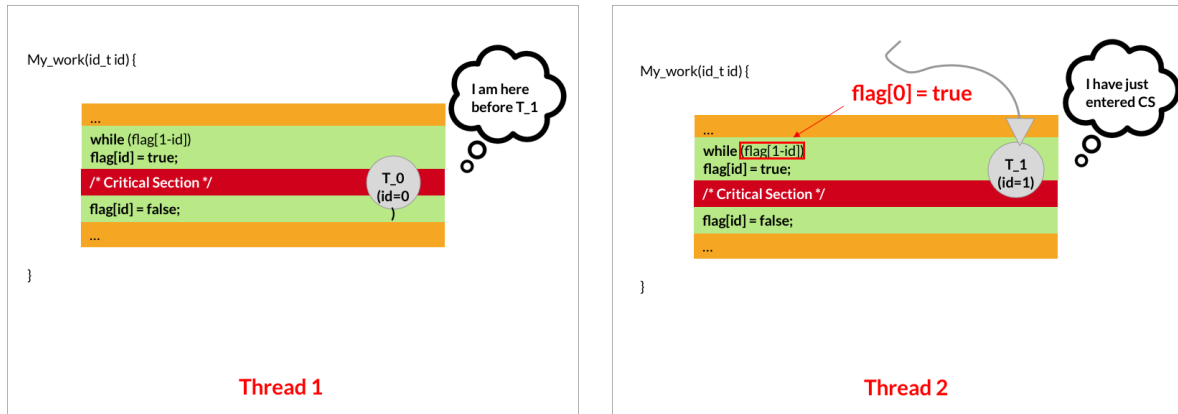
- CS can be entered only one thread at a time

BAD Progress is not satisfied

- A thread may not be able to enter CS, even if another is in the code section

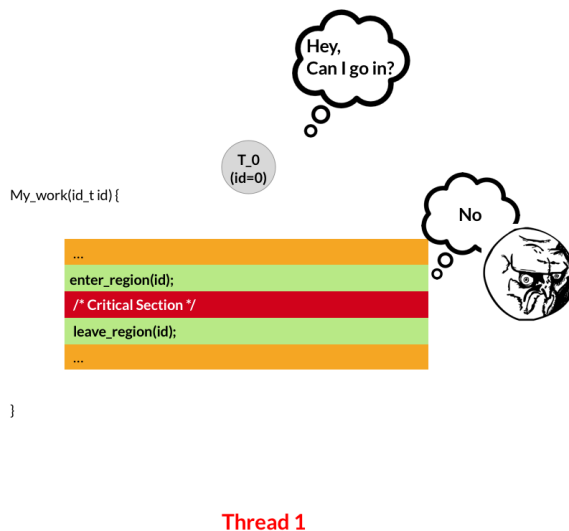


• 2-Thread Solutions: 2nd Try



- Peterson's Algorithm

- Is a solution to critical section problem
- Is developed by Gary L. Peterson in 1981
- Doesn't require any special hardware
- Allows critical section to be processed one thread at a time
 - * By rejecting other threads that tries to access the critical section at the same time



- Higher-Level Abstractions for CS's

- Are other solutions to critical section problem
 1. Lock
 2. Semaphores
 3. Messages

4. Monitors

- Will be mentioned in detail in week 3 notes
- Synchronization Hardware
- Atomic Instructions: Test-and-Set Lock (TSL)
 - Is a synchronization mechanism
 - * No other processes can begin until first is finished
 - Uses two variables
 - * $Lock == 0 \Rightarrow$ Nobody is using the lock
 - * $Lock == 1 \Rightarrow$ Lock is in use
 - Hardware executes the code with lock **atomically**
 - * **atomic** means without interruption
 - It's like a door lock used in portable bathroom.



- A Lock implementation

- Uses two methods

1. acquire

```
1 void acquire(boolean *lock) {  
2     while(text_and_set(lock)); //<- note the semicolon  
3     here :}  
4 }
```

2. release

```

1  void release(boolean *lock) {
2      *lock = false;
3  }
4

```

- Using Lock

Function Definitions

Withdraw(acct, amt) {

```

    acquire(lock);
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct, balance);
    release(lock);
    return balance;
}

```

Deposit(account, amount) {

```

    acquire(lock);
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct, balance);
    release(lock);
    return balance;
}

```

Loop

Possible schedule

```

acquire(lock);
balance = get_balance(acct);
balance = balance - amt;

```

```

acquire(lock);

```

```

put_balance(acct, balance);
release(lock);

```

```

balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
release(lock);

```