

Computer Science 534 – Operating Systems

Processes and Process Management

(Paraphrased from *Modern Operating Systems* by Andrew Tannenbaum)

- Most modern operating systems are capable of performing several tasks in what appears to be a simultaneous manner
- In fact, the CPU switches from program to program, each running for tens or hundreds of milliseconds at a time
- While at any given time the CPU is only working on a single program, in any given second, the CPU can be working on several programs
- This leads to the impression that the programs are running in parallel

The Process Model

- In the process model, all runnable software (and sometimes the OS) is organized into a number of sequential processes or processes for short
- A process is essentially an executing program that includes the current values of the program counter, registers, and variables
- Conceptually each process has its own CPU
- Because the CPU is switching rapidly between processes, the rate at which processes perform their tasks is not uniform
- This implies that no timing assumptions should be made when programming processes
- There is a difference between a program and a process however subtle it might be

Consider the following analogy:

- A baker is going to bake a new kind of cake
- The baker has a cake **recipe** and all the necessary ingredients for **input** into the cake: flour, eggs, milk, sugar, etc
- In this case, the **recipe** is the **program** (i.e., an algorithm expressed in notation suitable to be run on a computer)
- The **baker** is the **CPU**
- The cake **ingredients** are the **input data**

- The **process** is the activity of the **baker reading the recipe, fetching the ingredients, and baking the cake**
- Now, consider that the baker's assistant cuts his finger badly with a knife while the baker is in the middle of preparing the ingredients for the cake
- The baker makes a note of where she is in the recipe (the **state of the current process is saved**)
- The baker gets the first aid book and begins following directions for stopping the bleeding
- The baker (the **CPU**) has switched from a **LOWER PRIORITY** process (baking a cake) to a **HIGHER PRIORITY** process (making sure the assistant doesn't bleed to death)
- **Each of these activities has a program...recipe vs. first aid book**
- When the bleeding has stopped and the ambulance has taken the assistant to the hospital, the baker gets back to baking the cake, **picking up in the recipe where she left off.**
- **Key idea: a process is an activity of some kind...it has a program, input, output, and a state**
- In the PC world, a single processor is usually shared between processes with a **scheduling algorithm** used to determine when a process gets CPU time, and when it must give up the CPU to another process

Process Hierarchies

- UNIX provides a means by which all required processes be created
- The mechanism is the **fork()** system call
- fork() creates an **identical copy** of the calling process
- After the process is created, the new process (**child**) **runs in parallel** with the calling process (**parent**)
- Since parents and children can use fork() to create a new process, a hierarchical tree of processes can be created that is arbitrarily deep

Process States

- Processes often need to interact with other processes
- Given the command line:

```
$ cat file1 | grep tree
```
- The **first** process (**running cat**) reads a file
- The **second** process (**running grep**) selects all lines containing the word “tree” in the file
- Depending on many factors (speed and complexity of the two processes for example), **it might be the case that grep is ready to run, but there is no input waiting for it**
- The process running grep **cannot continue** because it has no input to process
- The process must change to a **blocking state** until input becomes available
- **When a process blocks, it does so because it can not logically continue**
- **There three possible states a process can be in:**
 1. **Running** (actually using the CPU at that instant)
 2. **Ready** (runnable; temporarily stopped to let another process run)
 3. **Blocked** (unable to run until some external event takes place)

There are four possible transitions between states:

Transition 1: Running to Blocked

- Occurs when a process discovers that it cannot continue

Transition 2: Running to Ready

- Caused by the scheduler without the “knowledge” of the process
- Occurs when the scheduler has determined that the process has had enough CPU time and turns the CPU over to another process

Transition 3: Ready to Running

- Again, caused by the scheduler without the “knowledge” of the process
- Occurs when all other processes have had their CPU time, and the scheduler turns the CPU over to the next process in turn

Transition 4: Blocked to Ready

- Occurs when the arrival of an external event for which the process was waiting (like a pipe waiting for input from a source) happens

Implementation of Processes

- The operating system maintains a **table** (an array of structures) called the ***process table***
- The process table contains ***one entry per process***

Each entry contains the following information:

- Process state information
- Process program counter
- Stack pointer
- Memory allocation
- Status of any open files
- Accounting and scheduling information
- Anything else about the process that must be saved when the process is switched from the running state to the ready state so that it can be **RESTARTED** later as if it never stopped running

The exact content of the process table varies with the operating system

The table can be generally divided into three main sections containing fields that deal with:

- **Process Management**
- **Memory Management**
- **File Management**

Some typical UNIX process table fields

Process Management	Memory Management	File Management
Registers	Pointer to text segment	UMASK mask
Program Counter	Pointer to data segment	Root directory
Program Status Word	Pointer to bss segment	Working directory
Stack Pointer	Exit status	File descriptors
Process State	Signal status	Effective uid
Time when process started	Process ID	Effective gid
Children's CPU time	Parent process	System call parameters
Time of next alarm	Process group	Various flag bits
Message queue pointers	Real uid	
Pending signal bits	Effective uid	
Process ID	Real gid	
Various flag bits	Effective gid	
	Bit maps for signals	
	Various flag bits	

The Illusion of Multiple Sequential Processes with One CPU and Many Devices

Interrupt Vector: A pointer to a hardware interrupt routine. The Interrupt Vector Table resides in low memory, the interrupt routines themselves reside elsewhere

Suppose that user process 3 is running when a disk interrupt occurs...what happens next? Three major steps are involved.

Step One

The following are pushed onto a stack by the interrupt hardware:

- **program counter**
- **program status word**
- **contents of one or more registers**

The computer jumps to the address pointed to by the interrupt vector and begins executing the interrupt service procedure

Step Two

- The interrupt service procedure saves all registers in the process table entry for the current process
- The current process number and the pointer to it are kept in global variables for easy and quick access

Step Two (continued)

- The information deposited by the interrupt is **popped off the stack** and the stack pointer is set to a temporary stack used by the process handler
- **Note that setting the stack pointer cannot be done with C or any other high level language...this is accomplished with small assembly language routines making them platform specific**

Step Three

- Determine which process started the disk request
- Normally the calling process will have gone to **SLEEP** after issuing the request so it must be awakened
- The state of the process is changed from blocked to ready
- The scheduler is called
- Whether this process is run next depends on the scheduling algorithm
- There are at least two processes ready to go:
 - **the process that started the disk I/O**
 - **the process that was interrupted**
- UNIX gives higher priority to **I/O-bound processes** than CPU-bound processes
- This strategy maximizes the amount of parallelism keeping both the CPU and the disk busy
- Other systems emphasize different schemes, e.g., fairness over efficiency
- The C procedure called by the assembly language interrupt code returns
- The assembly language code loads the registers and memory map for the now-current process and starts it running

Skeleton of an OS Interrupt Operation

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector
3. Assembly language routine saves CPU registers
4. Assembly language routine sets up new stack
5. C routine marks service as ready
6. Scheduler decides which process to run next
7. C routine returns to assembly routine
8. Assembly language routine starts up new process

Interprocess Communication

- **Interprocess communications (IPC)** provide a way for processes to pass information to each other, **preferably without the use of interrupts**
- **Example:** Process One sends output through a pipe to the input of Process Two
- In UNIX, some processes that work together share common storage space to which they can read and write
- The space could be memory, a file, etc.
- There are some problems that are associated with this kind of resource sharing

Race Conditions

Consider the following the case of the **Print Spooler**:

- When a process wants to print a file, it enters the filename in a special spooler directory
- Another process, the Spooler Daemon, periodically checks to see if there are any files to be printed, prints them, then removes their names from the directory

Race Conditions (continued)

- Suppose that the spooler directory has a large number of slots numbered 0 through n, each capable of holding a file name

Also, suppose that there are two shared variables:

- **out**: points to the next file to be printed
- **in**: points to the next free slot in the directory

At **some certain instant**, slot 0 and slot 3 are **empty** (files have been printed)

At the same instant, slot 4 and slot 6 are **full** (holding filenames queued for printing)

More or less simultaneously, Process A and Process B decide they want to queue a file to be printed

Where Murphy's Law applies, the following is a possible scenario:

1. **Process A reads** the value in the variable ***in*** and stores the value **7** in a local variable called ***next_free_slot***
2. Just then a **clock interrupt occurs** and the **scheduler** decides that Process A has run long enough and switches to Process B
3. **Process B reads** the value in the variable ***in*** and also gets a **7**
4. **Process B stores** the name of its file in **slot 7** then **updates it to 8**
5. **Process B** then goes off to get a burger
6. Eventually, **Process A** gets another turn and **starts up where it left off**
7. **Process A looks in *next_free_slot*** and finds a 7 and **writes** its filename in **slot 7**
8. The filename that **Process B placed in slot 7 is overwritten**
9. **Process A** computes ***next_free_slot+1*** (which is 8) and sets **in** to 8
10. The spooler directory is **internally consistent** so the **printer daemon will not notice anything wrong**
11. Unfortunately, **Process B will never get any printed output**
12. These timing situations are likened to people **racing after the same thing at the same time**

Critical Sections

- **A Critical Section is that part of a program where a shared resource is accessed by a process**
- Critical Sections provide a way around Race Conditions and other ***shared access problems***
- Critical Sections provide a means of **excluding** any process from reading from and writing to a shared resource while another process is using it
- In other words, we seek **MUTUAL EXCLUSION**...a way of making sure that if **one process is using a shared resource, all other processes will be excluded from doing the same thing AT THE SAME TIME**
- Achieving mutual exclusion is a **major issue** in operating system design

There are four conditions that must be in place to provide mutual exclusion:

1. **No two processes may simultaneously be inside their critical sections**
2. **No assumptions may be made about speeds or the number of CPUs**
3. **No process running outside its critical section may block other processes**
4. **No process should have to wait forever to enter its critical section**

Strategies For Achieving Mutual Exclusion

Disabling Interrupts

- **Disabling interrupts** will disable the **clock interrupt** so the scheduler will **not switch processes** until the clock interrupt is enabled again
- Disabling interrupts is **sometimes a useful technique within the kernel**, but it is **not appropriate** to let **user processes disable interrupts...they might never turn them back on**

Lock Variables

- A software solution
- Consider having a **single shared variable called *lock*** that is initialized to **zero**
- When a process wants to **enter its critical section** it tests the variable
- **If *lock* = 0, the process sets it to 1 and enters its critical section**
- **If *lock* = 1, the process waits until *lock* = 0**
- **Thus, when *lock* = 0, no process is in its critical section**
- **When *lock* = 1, some process is in its critical section**
- **This solution suffers from the fatal race condition flaw:**
- **Suppose one process reads the lock and sees that it is zero**
- **Before it can set the lock to one, the scheduler switches to another process**
- **The now running process sets the lock to 1**
- **When the first process runs again, it will also set the lock to one, and the two processes will be in their critical sections at the same time**
- **Checking the lock before entering, then checking again before writing will do no good, as the same scheduling conflicts can occur no matter how many times the lock is checked**

Strict Alternation

- Strict alternation insists that **each process get a turn in its critical section** by checking a variable for zero.
- If the value is zero, the process enters its critical section while the other process spins in a holding pattern, continuously testing the variable, doing nothing until it is allowed to go to its critical section when the variable is zero again

Strict Alternation (continued)

- While this appears to be reasonable, there are **serious problems** with this strategy
- Continuously testing a variable waiting for a specific value to appear is called **busy waiting, and seriously wastes CPU time**
- **Busy waiting should be avoided**
- Other issues arise when one process is much slower or much faster than the other
- This condition can lead to a **process being blocked** by another process that is **NOT in its critical section**

Peterson's Solution

- An initial software solution to the mutual exclusion problem was devised by the Dutch mathematician, **T. Dekker** in the late 1950s or early 1960s
- Dekker's solution involved the use of lock variables and warning variables and didn't require strict alternation
- **G.L. Peterson** discovered a much simpler solution in **1981 rendering Dekker's solution obsolete**
- Peterson's solution consists of **two functions: enter_region (int process), and leave_region (int process)**
- Before entering its critical region, a process calls **enter_region** passing its **own process number, 0, or 1 as a parameter**
- This will cause it to **wait** (if need be) **until safe to enter**
- After the process is finished with the shared resources, the process calls **leave_region** to indicate that it is done
- This allows another process to enter its critical process if desired
- While Peterson's solution **does enforce mutual exclusion**, it also suffers from enforcing **busy waiting**...a process sits in a tight waiting loop until it is safe to enter its critical section

The Priority Inversion Problem

- Consider a high priority process, **H**, and a low priority process, **L** share resources
- The scheduling rate is such that **H** is run whenever it is in a ready state
- At a certain moment when **L** is in its critical section, **H** becomes ready to run
- **H** now begins **busy waiting**, but since **L** is **never scheduled** while **H** is running, **L** never gets a chance to leave its critical section and **H** loops forever

SLEEP and WAKEUP

- There are some IPC constructs that **block** instead of **cause busy waiting** when they are not allowed to enter their critical sections
- One of the simplest is the **SLEEP/WAKEUP** pair
- **SLEEP** is a system call that causes a process to be suspended (**block**) until another process wakes it up
- The **WAKEUP** call has one parameter: **the process to be awakened**

The Producer-Consumer Problem

Assume that two process share a **common, fixed-size memory buffer**

One process (the **producer**) **puts data into the buffer**, and the other process (the **consumer**) **removes data from the buffer**

Trouble arises when the **producer wants to put new data into the buffer, but the buffer is full**

The solution is for the **producer to go to sleep**, then be awakened when the consumer has removed one or more items

Similarly, if the **consumer wants to remove items but the buffer is empty**, the consumer goes to sleep and is awakened when the producer places something in the buffer

There is a potentially serious problem with this solution

Sleep and Wakeup (continued)

- **Given:** N is the **maximum number of items** that can be placed in the buffer
- **Given:** a common variable in the producer and consumer code called ***count***
- The **producer** will go to sleep when ***count = N*** (there is nowhere to put anything)
- The **producer** will wake up when something has been removed by the consumer, and the consumer has decremented ***count*** indicating that there is now space to put a new item
- The **consumer** will go to sleep when ***count = 0*** (the buffer is empty...nothing to consume)
- The **consumer** will wake up when something is placed in the buffer by the producer (***count will then be > 0***...the buffer has items to consume)
- **Consider the following scenario:**
- The buffer is **empty** and **consumer has just read count to see if it zero**
- At that instant, the **scheduler** decides to stop running the consumer and start running the producer
- The producer enters an item into the buffer, increments count, and notices that count is 1
- Reasoning that count was zero before the increment, the producer assumes that the **consumer must be sleeping and issues a WAKEUP call to the consumer**
- **Unfortunately**, the **consumer was not logically asleep** so the **wakeup signal IS LOST**
- When the consumer runs next, it will **test the value of count it PREVIOUSLY read, find it to be zero, and go to sleep**
- Sooner or later, **the producer will fill the buffer to N items, and will then go to sleep**

- ***They will both sleep forever***
- **There are some quick fixes to the problem that include adding check bits for sleep and awake status, but eventually, this strategy will fail too**

Semaphores

- In 1965, **E. W. Dijkstra** suggested the use of an integer variable to count the number of wakeups saved for future use
- In his paper, he proposed a new variable type called a **semaphore**
- A **semaphore** would have the value of **zero** if there were **no saved wakeups** and some **positive value** if there were **one or more saved wakeups**
- Dijkstra also proposed two operations, **DOWN** and **UP** which are generalizations of **SLEEP** and **WAKEUP** respectively
- The **DOWN** operation on a semaphore **checks to see if the value is greater than zero**
- If so, it decrements the value and continues
- If the value is zero, the **process is put to sleep**
- **Checking the value, changing the value, and possibly going to sleep are all done as a *single, indivisible atomic action***
- This **guarantees** that once a semaphore operation has started, **NO OTHER PROCESS CAN ACCESS THE SEMAPHORE UNTIL THE OPERATION IS COMPLETED OR BLOCKED**
- This “**atomicity**” is **absolutely essential** to solving synchronization problems and avoiding race conditions
- The **UP** operation increments the value of the semaphore addressed
- If one or more processes were sleeping on that semaphore (unable to complete a DOWN operation), **one is chosen by the system to complete its DOWN**

- After an **UP** operation on a semaphore with processes sleeping on it, the **semaphore will still be at zero**...there will just be one less process sleeping on it
- **The operation of incrementing the semaphore and waking up one process is also indivisible**
- **No process ever blocks during an UP as no process ever blocks doing a WAKEUP in the previous model**

Solution to the Producer – Consumer Problem Using Semaphores

```

#include "prototypes.h"

#define N 100                /* number of slots in the buffer */

typedef int semaphore;      /* semaphores are a special kind of int */

semaphore mutex = 1;        /* controls access to critical region */
semaphore empty = N;        /* counts empty buffer slots */
semaphore full = 0;         /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {          /* TRUE is the constant 1 */
        produce_item(&item); /* generate something to put in buffer */
        down(&empty);         /* decrement empty count */
        down(&mutex);         /* enter critical region */
        enter_item(item);     /* put new item in buffer */
        up(&mutex);           /* leave critical region */
        up(&full);            /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {          /* infinite loop */
        down(&full);           /* decrement full count */
        down(&mutex);         /* enter critical region */
        remove_item(&item);  /* take item from buffer */
        up(&mutex);           /* leave critical region */
        up(&empty);           /* increment count of empty slots */
        consume_item(item);  /* do something with the item */
    }
}

```

Semaphores (continued)

- The ***mutex semaphore*** is used to assure **mutual exclusion**
- **This guarantees that only one process at a time will be reading from or writing to the buffer and associated variables**
- The ***full*** and ***empty semaphores*** are used for **synchronization...to guarantee that a certain event does or does not occur**
- In the this case, ***full*** assures that the **producer will stop producing** when the buffer is full, and ***empty*** assures that the **consumer stops when the buffer is empty**

Other Constructs

There are other methods of solving the problems associated with the producer – consumer problem

Event Counters

- Counter variables are used to keep a running total of the **items produced and the items consumed**
- **Using event counters**, the producer consumer problem can be solved **without the use of mutual exclusion**

Monitors

- A monitor is a collection of procedures, variables and data structures that are placed together in a special kind of package or module
- **Processes may call the procedures in a monitor whenever they want**
- **Procedures cannot access the monitor's internal data structures from procedures declared OUTSIDE the monitor**
- Does this sound something like an **OBJECT**?
- Monitors have an important property that allows them to ensure mutual exclusion: **only ONE process can be active in a monitor at any instant**
- It is important to note that monitors are programming language concepts
- **Most languages (C, Pascal, ...) do not have monitors so the compilers cannot enforce any rules of mutual exclusion**

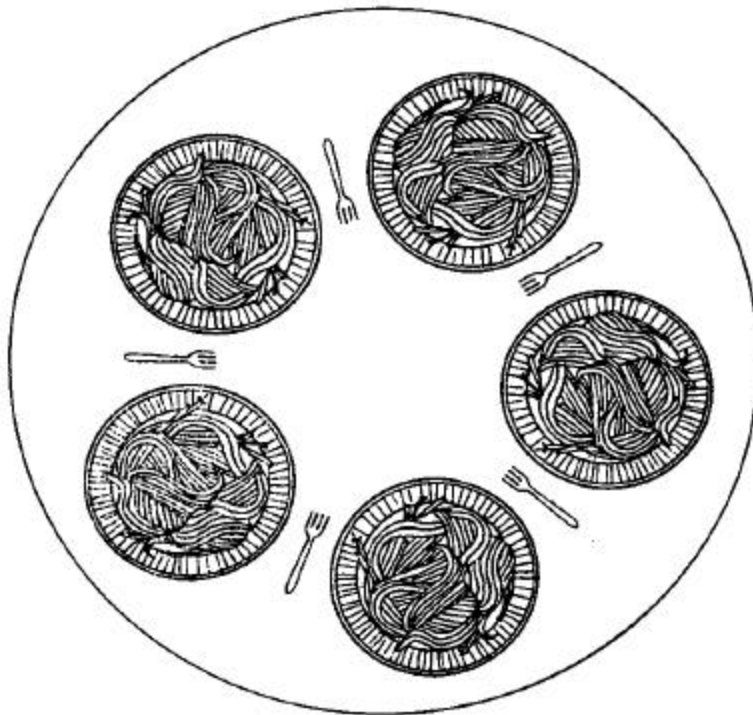
A Classic IPC Problem

The Dining Philosophers Problem

In 1965, **Dijkstra** posed and solved a **synchronization** problem called the *Dining Philosophers Problem*

The problem is stated as follows:

- Five philosophers are seated around a table
- Each philosopher has a plate of spaghetti
- The spaghetti is so slippery that a philosopher needs two forks to eat it
- Between each plate is a fork
- The table is shown below:



Dining Philosophers (continued)

- The philosopher's life consists of alternating periods of eating and thinking
- When a philosopher gets hungry they try to acquire their right and left forks, one at a time, in either order
- If successful in acquiring two forks, they eat for a while, put down the forks, and continue to think

The question:

Can a program be written for each philosopher, that does what it is supposed to do, without the computer getting stuck?

The Answer: Yes

Two Other Major Problems

Deadlock

- A set of processes is **deadlocked** if each process in the set is waiting for an event that only another process in the set can cause
- Or, each member of the set of deadlocked processes is waiting for a resource owned by a deadlocked process

Starvation

- A situation where all processes continue to run indefinitely but fail to make any progress
- An example might be each philosopher picking up their right fork only to find that there are no forks...so they all put down their right forks, and then they all pick up their left forks...over and over and over...

Process Scheduling

The *scheduler* decides which process is going to get the CPU next

There are several criteria that a good scheduling algorithm must take into account:

Fairness

- Make sure that each process gets its fair share of the CPU

Efficiency

- Keep the CPU busy 100% of the time

Response Time

- Minimize response time for interactive users

Turnaround Time

- Minimize the time that batch users wait for output

Throughput

- Maximize the number of jobs processed per hour

Some of these tenets are contradictory

For example, to **minimize response time for interactive users**, the scheduler **shouldn't run any batch jobs except in the wee hours of the morning**

It can be shown that a scheduling algorithm that favors any class of jobs hurts another

Such is life

There are two types of scheduling algorithms:

Preemptive Scheduling

- Allow processes that are logically able to run to be temporarily suspended in favor of another process

Nonpreemptive Scheduling (Run to Completion)

- Not a good general purpose solution on multiuser machines
- **Consider a colleague wanting to calculate pi to 100 billion places...**

Typical Scheduling Algorithms

Round Robin Scheduling

- One of the oldest, fairest, and most widely used algorithms
- Each process is assigned a time interval called its *quantum*
- This is the amount of time it is *allowed to run*
- If the process is still running at the end of the quantum, the scheduler *preempts* and gives the CPU to the next process
- If the process is blocked or has finished its work before the end of its quantum, the switching is done when the process blocks (or has finished)
- The only real issue in round robin is the *length of the quantum*
- Note that switching to another process requires that data be saved
- This overhead takes time
- Switching between processes is called *context switching*
- Suppose that a context switch takes 5mS
- Suppose that the quantum is set at 20mS
- After doing 20mS of work, the CPU will spend 5mS switching contexts
- This represents 20% of the CPU's time wasted on administrative tasks
- Setting the quantum to a longer period (say 500mS) would cause inordinate delays to interactive users at the tail end of the queue
- Consider that if ten users hit the return key at approximately the same time, ten jobs got into the scheduling queue
- If the queue is empty, the first job starts immediately, the next ½ second later, the next, 1 second later, and so on
- The 10th user will have to wait 5 seconds for a response from the system
- **Not good**

Round Robin (continued)

The conclusion:

- Setting the quantum **too short** causes **too many context switches** and wastes a large percentage of CPU time on administrative overhead
- Setting the quantum **too large** will cause **poor response to interactive requests for service**
- A quantum of 100mS appears to be a reasonable compromise

Priority Scheduling

- Round Robin **assumes implicitly that all jobs have equal priority**
- ***In the real world, this is not the situation***
- For example, at a university computing center, the **Deans** might have first priority (**even though they probably are the laziest users** on campus), then professors, secretaries, custodians, **then students**
- This leads to the necessity of an scheduling algorithm that can **prioritize jobs**
- The basic idea: each job is assigned a priority and the process that is ready to run having the highest priority gets the CPU
- To prevent high priority jobs from never letting go of the CPU, the scheduler might **decrease the priority of the process with every system clock tick (at each clock interrupt)**
- If the priority of a process under these circumstances **drops below that of the next highest process, a context switch is made to that process**
- **UNIX** has a command called **nice** that allows users to **voluntarily** reduce the priority of their processes...to be **NICE** to other users and accommodate their jobs
- **This utility is RARELY if EVER used**
- Typically, processes are **grouped into priority classes**
- **Priority scheduling is used among the classes, and round robin within each class**

Shortest Job First

- The previous algorithms were designed to **accommodate interactive systems**
- **Shortest Job First is designed for *optimizing batch jobs***
- **Assume an insurance company is processing the same kinds of claims everyday, and can accurately predict the processing time of a particular type of job**
- **Given the following:**

Job A: 8 units	Job B: 4 units	Job C: 4 units	Job D: 4 units
----------------	----------------	----------------	----------------

- Running the jobs will produce a **total run time of 20 units**
- The formula $(4A + 3B + 2C + D)/4$ will produce the **average turnaround time**
- **The turnaround time for A is 8 units, B is 12 units, C is 16 units and D is 20 units**
- **The average turnaround time is > 14 units**

Now, run the shortest job first

Job B: 4 units	Job C: 4 units	Job D: 4 units	Job A: 8 units
----------------	----------------	----------------	----------------

- The turnaround times are now **4, 8, 12, and 20 units respectively** for an **average of < 10 units**
- This demonstrates that the **first job is going to contribute most to the average**
- By placing the **shortest job first, (and the rest in ascending order of length) the average turnaround time can be reduced**
- **This can be proved for any number of jobs**
- **For batch jobs, Shortest Job First is optimal**