z/OS JCL and Utilities

z/OS JCL and Utilities - Course Objectives

On successful completion of this course, the student, with the aid of the appropriate reference materials, should be able to:

1. Understand the basic flow of work in z/OS, including JES Readers, Writers, Initiators, the role of the Interpreter, and the purpose of Allocation

2. Describe the storage layout of z/OS and use the REGION and MEMLIMIT parameters as appropriate and necessary

3. Code JCL statements as necessary to accomplish work in the z/OS environment, including JOB, EXEC, DD, OUTPUT, IF/THEN, ELSE, ENDIF, INCLUDE, SET, JCLLIB, PROC and PEND statements

4. Create and delete data sets using IEFBR14

5. Copy files for backup, restore, and testing purposes using the IBM utility program IEBGENER

6. Use some of the basic services of IDCAMS, the VSAM utility

7. Use a Sort/Merge program product to sort a sequential data set

8. Code the OUTPUT JCL statement to produce multiple groups of SYSOUT files

9. Use ISPF/PDF 3.8 and / or SDSF, OMC-FLASH, IOF, or (E)JES facilities for tracking jobs and examining job output (as available to the student)

10. Code cataloged procedures, including the use of symbolic parameters and defaults, nested procedures, and private proclibs

11. Describe the implications of Storage Management Subsystem (SMS) and Partitioned Data Sets, Extended (PDSE's)

12. Know where to find additional information as needed.

z/OS JCL and Utilities - Topical Outline

Day One

z/OS JCL and Utilities - Topical Outline, 2

Day Two

Day Three

z/OS JCL and Utilities - Topical Outline, 3

Day Three, continued

Section Preview

- Introduction
  - Operating System
  - The Application Program Environment
  - MVS - Multiple Virtual Storages
  - The Road to z/OS
  - z/OS Work Flow
  - Job Entry Subsystem
  - JCL Statement Format
  - JOB Statement Format
  - EXEC Statement Format
  - OSWTO (Machine Exercise)
  - JCL Clues - 1

Operating System

❒ **A Collection of programs that:**

Manage a computer system's resources

- **Maximize device utilization**
- **Transfer data between memory and devices at program request**
- **Handle error detection and recovery**
- **Attain maximum possible performance under current workload**

Schedule work to be done

- **Determine jobs to be run, based on job control statements**
- **Assign (allocate) resources to programs as necessary**
- **Handle unscheduled work such as time sharing systems and transaction processing work**
- **Communicate with operator via**
  Commands (operator to system)
  Messages (system to operator)

Maintain integrity of system and data

- **Provide security**
- **Prevent simultaneous update**
- **Prevent deadlock**

The Application Program Environment

❒ **An operating system provides an environment, a context, for application programs to run**

Control blocks keep track of all programs in memory, their location, attributes, and status

System services allow application programs to do I/O, manage memory dynamically, handle application errors, and much more

❒ **The most meaningful perspective here is how memory is organized, and we explore this in the following pages**

To show how memory is organized and, briefly, why it is organized that way

We do this by looking at a short history of MVS, MVS/XA, OS/390, then z/OS

✗ Roughly corresponding to addressing limits of 24-bits (MVS), 31-bits (MVS/XA and OS/390), and 64-bits (z/OS): the size of memory the hardware and software support

MVS - Multiple Virtual Storages

❒ **An operating system that runs on S/370 and later IBM mainframes**

Virtual Storage - the functional illusion of computer internal memory (storage), created using real internal memory, disk as a backing store, and hardware features of the CPU to map virtual addresses to real addresses

✗ Only the portions of virtual memory holding data and instructions currently being used need to be in real memory at any point in time

Address Space - a virtual storage that appears to be as large as the hardware addressing scheme allows (24-bit addresses, which allow up to 16MB of virtual storage per address space)

✗ Contains operating system code and user code

✗ Contains data currently being processed

Each user has their own, distinct Address Space

✗ System Address Spaces

✗ Batch Jobs

✗ Time Sharing Users

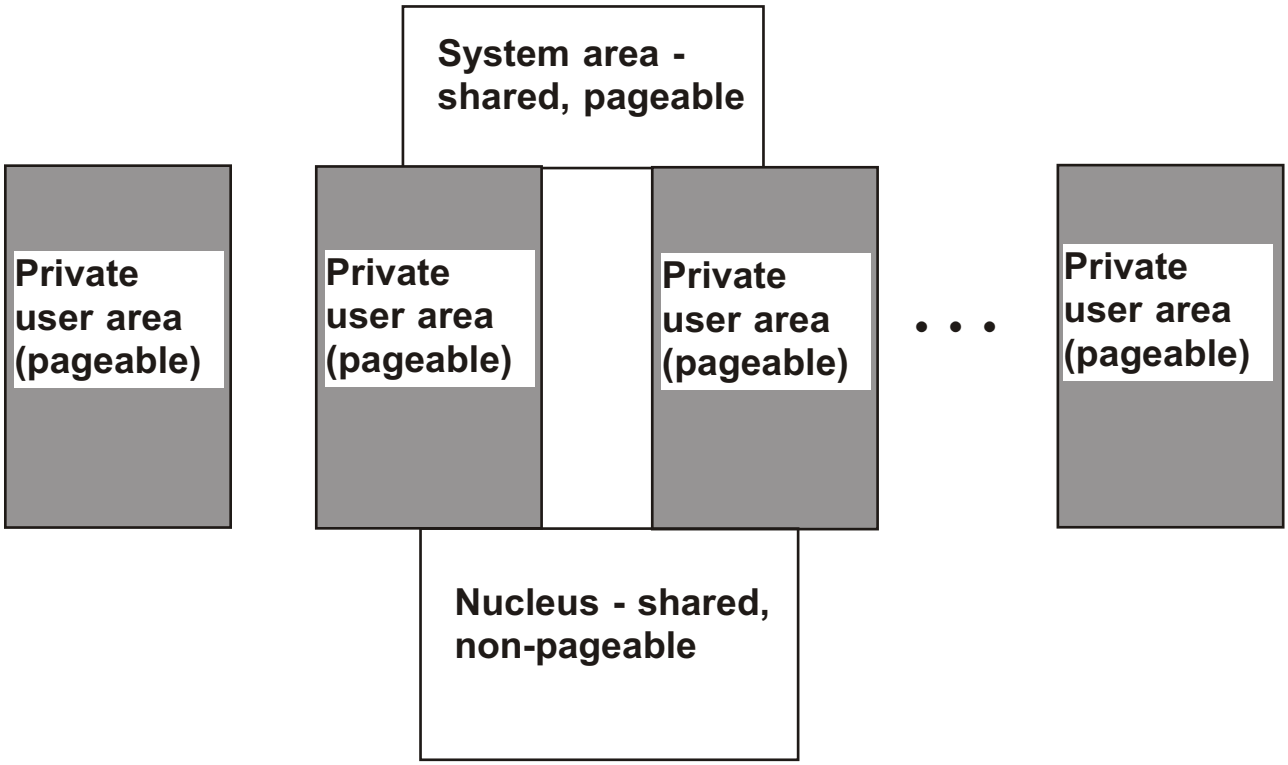
✗ Maximum of 32,767 Address Spaces total

The Road to z/OS

❒ **Because each user has their own address space, each address space needs to have a copy of the operating system**

Since this is the same for each user, the addressing scheme is set up to have only one, shared, copy of the nucleus area and the system area

The unique parts of an MVS system look, conceptually, like this:



❒ **Again, addresses are 24-bits so each address space is 16MB in size**

The Road to z/OS, 2

❒ **In the 1980's, IBM bit the bullet and extended the address space from 24 bits to 31 bits**

31 bits instead of 32 bits for a variety of reasons, which provides for a 2 GB address space (2,147,483,648 bytes)

This was called extended architecture, abbreviated XA, so the operating system was called MVS/XA

This provides for 128 times the previous amount of virtual storage for programs to use

In addition to providing a larger address space, IBM re-arranged the layout

✗ Sections of code that relied on 24-bit addresses had to remain under the 16 MB limit (which has come to be called The Line)

✗ So IBM moved as much of their code as possible above The Line (there will always have to be some code below The Line, to support older code)

So, the layout of an address space in MVS/XA looks like the diagram on the following page ...

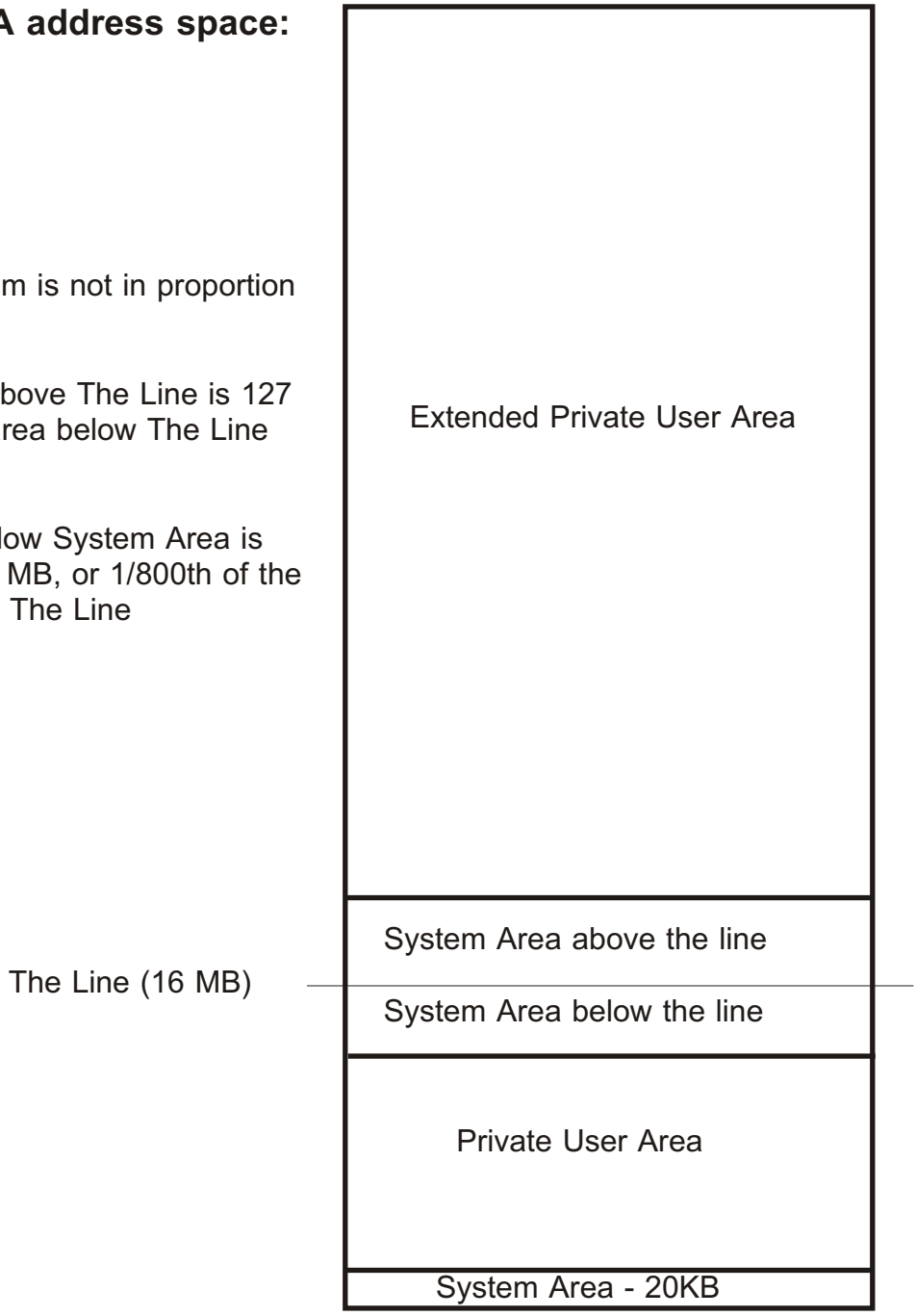
The Road to z/OS, 3

❒ **MVS/XA address space:**

This diagram is not in proportion

The area above The Line is 127 times the area below The Line

The 20KB low System Area is 1/50th of 1 MB, or 1/800th of the area below The Line



❒ **The goal is to put very little code and data below the line and to have the vast majority of programs and data reside above the line**

The Road to z/OS, 4

- **Other variations of MVS came along, to support enhanced hardware instructions and features, but the essence of address spaces did not change**

- **The next step in the evolution was OS/390 (Operating System/390) which is really a packaging of components**

- **OS/390 contains**

  MVS code plus a number of program products as a single package

  Intent was to update every six months, keeping all the products in synch, thus simplifying the process of installing and upgrading systems and products (1st release was 3/96)

  Products included with OS/390 (among others):

  - ✗ SMP/E (for maintenance uses)
  - ✗ TSO/E
  - ✗ ISPF
  - ✗ High Level Assembler
  - ✗ BookManager
  - ✗ DFSMSdfp
  - ✗ Language Environment (LE)
  - ✗ TCP/IP
  - ✗ DCE (Distributed Computing Environment support)
  - ✗ OpenEdition / POSIX support (UNIX under MVS!)

  In addition, other optional products are available to be shipped in an OS/390 order, for an extra charge

The Road to z/OS, 5

- **In 2001, IBM made available new hardware, the first of the zSeries machines, that supported 64-bit addresses**

  So now address spaces can be as large as 64-bit addresses allow

- **A new operating system, z/OS, was announced to support the new hardware**

- **But z/OS is based on OS/390 - there is a solid continuity here**

  Most old code can still run under z/OS, even code compiled and linked under earlier operating systems over 35 years earlier

  To use new features, of course, you need to rewrite, recompile, and rebind

  There are still address spaces, just larger and organized slightly differently

  There is still an MVS component, a TSO component, and so on

- **The last release of OS/390 was V2R10, available September 2000, the first release of z/OS was available March 2001**

  The announced intent is to slow the release schedule to once a year after V1R6 is available

The Road to z/OS, 6

❒ **Some of the issues around establishing a 64-bit address space are resolved this way**

The size of the low System Area is increased to 24KB

The previous limit of 2 GB is now called <u>The Bar</u>

✗ So programs or data can reside

➢ Below The Line

➢ Above The Line but below The Bar

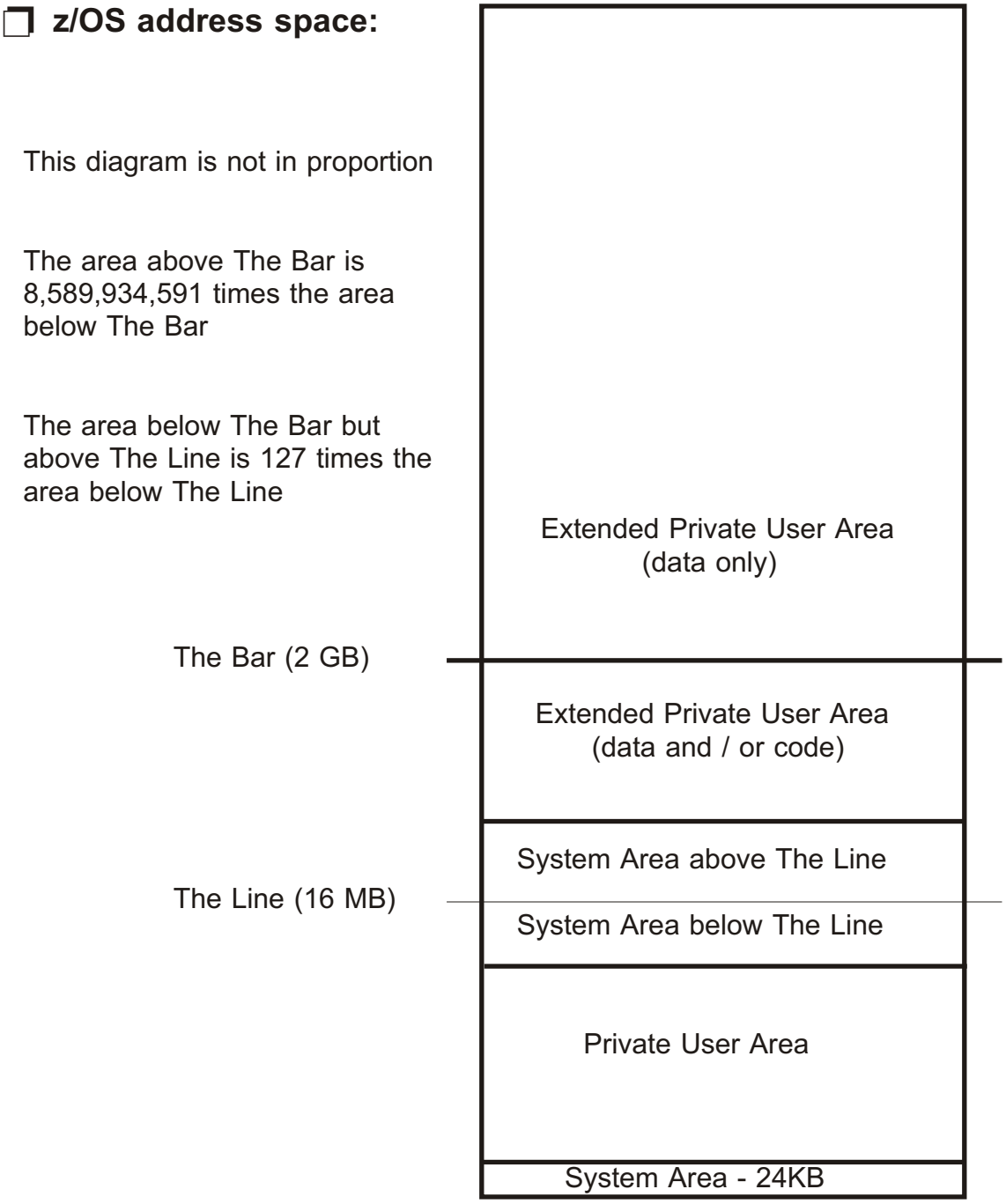
➢ Above The Bar (data only, currently, no programs)

❒ **A 64-bit address space allows for a maximum address of 18,446,744,073,709,551,615**

That is, a 64-bit address space is 8,589,934,592 <u>times</u> the size of a 31-bit, 2 GB address space

The Road to z/OS, 7

❒ **z/OS address space:**



❒ **Each job runs in its own address space, so now we move on to explore the management of jobs in z/OS ...**

Job Management

Job

- ✗ A unit of work to be run in the batch; one or more programs to be run in sequence

Job Queue

- ✗ An ordered collection of jobs

Job Class

- ✗ A one character code (A-Z, 0-9; 36 possibilities) assigned to each job

Job Priority

- ✗ A numeric value, 1-15 (1-13 for JES3 environment), that describes the relative importance of jobs within their job class (the higher the job priority number, the more important the job)

Job Management, 2

Job Control Language (JCL)

- ✗ A specification language used to describe jobs (work to be done) in terms of what resources are required, in what order, and under what conditions various work should get done

- ✗ JCL is written as a series of statements

Job Stream

- ✗ A collection of JCL and card [-image] input data read into the system for placement on the job queue

SPOOL

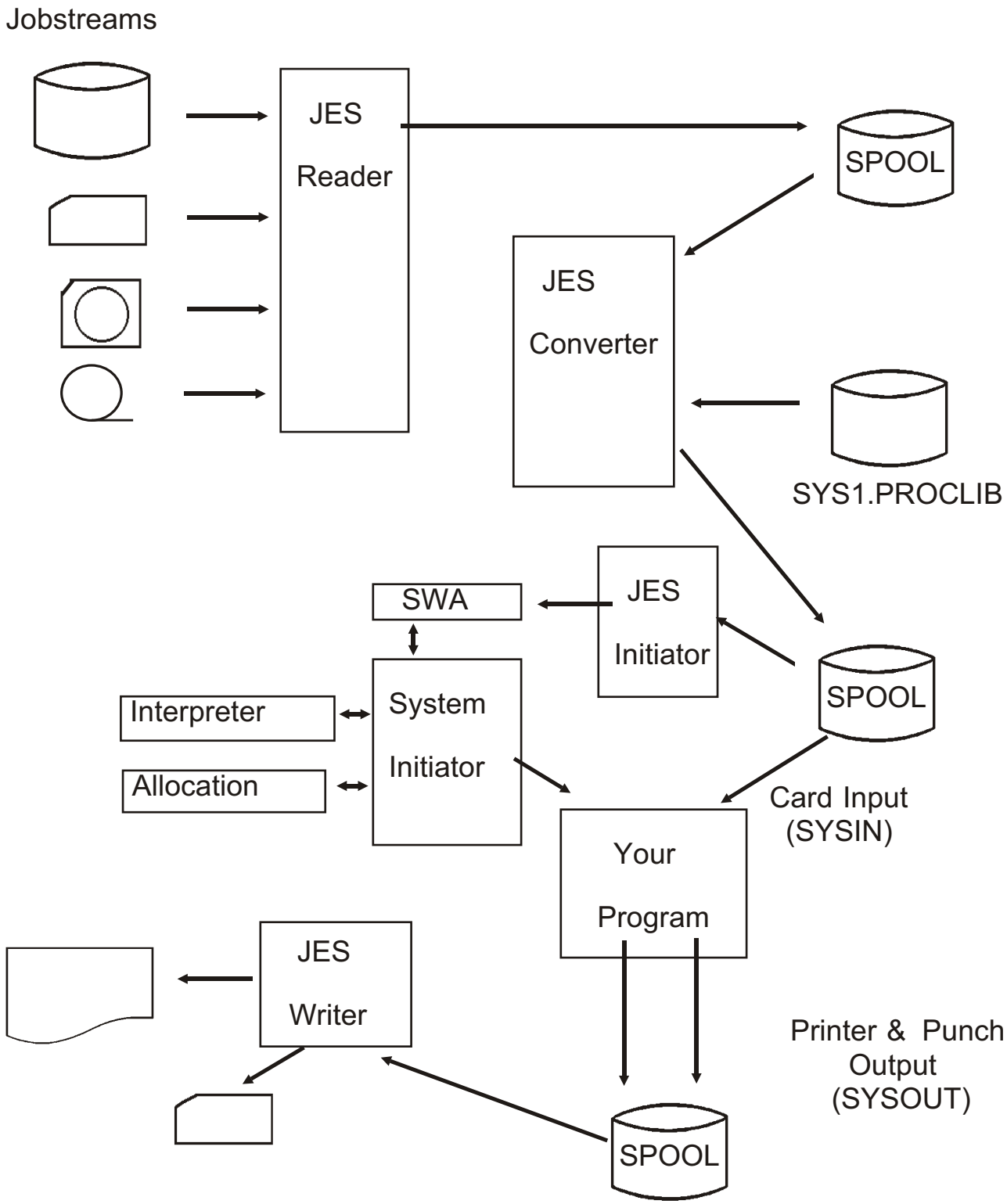
- ✗ Simultaneous Peripheral Operations On-Line

  - ➢ Using DASD work space to simulate the presence of multiple card readers, card punches, and printers

  - ➢ Thus allowing many jobs to be producing reports concurrently, even if you only have one printer

- ✗ The SPOOL area of DASD is also used to hold jobs in the queue waiting for work ...

z/OS Work Flow



Job Entry Subsystem (JES)

Reader

Reads job stream, puts on Job Queue by priority within class

Converter

Converts free form JCL into control blocks on Job Queue

Initiator

Selects which job to run next, based on class and priority

Allocation

Creates the required environment for executing the job

While the program runs, the SPOOL routines

Replace unit record I/O requests with I/O to SPOOL

Deallocation

Frees resources on completion of step and job

Writer

Transcribes SPOOLed output to printer or punch

Job Purge Routine

- ❒ **When the last line has been printed and the last card punched, the purge routine is invoked**

  Removes job JCL, and all SYSIN-type and SYSOUT-type records from SPOOL

  Frees that SPOOL space to be used by subsequent jobs

- ❒ **Note that there are two versions of JES: JES2 and JES3**

  The differences need not concern us here

  Also note: JES3 support will be going away

  - ✗ **Phoenix Software has arranged for a license of the source code and they are creating a product to continue and enhance JES3 support under their own product line**

How JCL Describes Resources

```
//jobname          JOB     (accountnginfo),prgrmrname,TIME=(min,sec),CLASS=x
//STEP1            EXEC        PGM=ISDED01,PARM='YNOFOUT/'
//STEPLIB            DD        DSN=DFIR.PROD.LOADLIB,DISP=SHR
//TRANSIN            DD        *
3560199227768
3568834990022                          sysin-type data (instream data)
4492445502367
•
•
•
//TRANSOUT           DD        UNIT=SYSDA,DISP=(,PASS),SPACE=(TRK,(25,10))
//MASTREF            DD        DSN=DFIR.CUST.MASTRFLE,DISP=SHR
//STEP2            EXEC        PGM=ISDUPDT,PARM='TEST/'
//GOODTRAN           DD        DSN=*.STEP1.TRANSOUT,DISP=(OLD,DELETE)
//MAST               DD        DSN=DFIR.CUST.MASTRFLE,DISP=OLD
//LOGTPE             DD        DSN=DFIR.APLILOG(+1),UNIT=TAPE,DISP=(,CATLG)
//LOGRPT             DD        SYSOUT=M
//UPDRPT             DD        SYSOUT=A,COPIES=2
//SYSUDUMP           DD        SYSOUT=D
```

The Allocation Process

Job Allocation

Step Allocation

- ✗ Units, volumes, data sets, DASD space; then program fetch loads in the program

STEP EXECUTION

Step Deallocation

- ✗ Data set disposition processing

Job Deallocation

Final data set dispositions

Indicate SYSOUT-type data available for processing

JCL Operations

- **JCL describes the work to be done in a job through statements that are categorized into operations**

- **There are four major JCL operations, each described in separate JCL statements:**

  JOB statement

  Indicate the start of the JCL for a job; assign job class (which initiators can service this job), and some other basic descriptive information

  EXEC statement

  Indicate step boundaries; each step runs one program

  DD statement

  Data Definition; one for each data set resource the program in a step will need

  OUTPUT statement

  Describe SYSOUT-type processing characteristics for some data sets

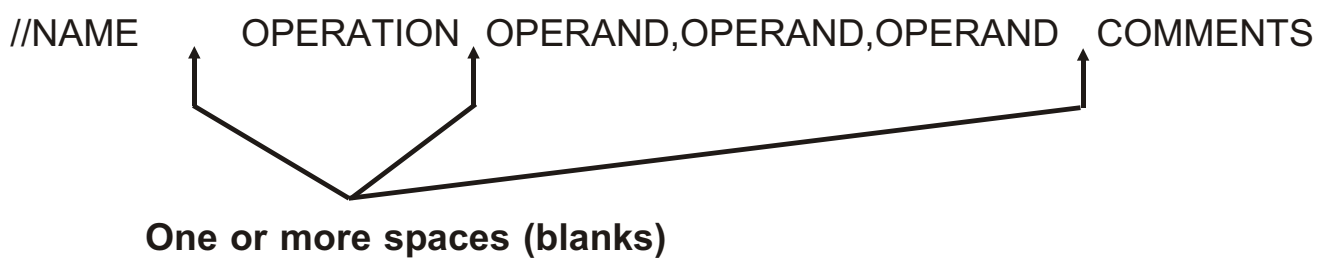
- **Every job has one JOB statement followed by an EXEC statement for each program to run**

- **Each EXEC statement is followed by the DD statements that describe the data sets the program run in that step will use**

- **OUTPUT statements and their placement are described later**

JCL Statement Format

Columns 1-71:



- **NAME: 1-8 characters from A-Z, 0-9, $, #, @; first not numeric**

Operation	Name field
JOB	jobname
EXEC	stepname
DD	ddname
OUTPUT	outputname

- **OPERANDS: Positional,Keyword**

JCL Statement Format, 2

- **Special formats:**

//*	**—**	**Comment statement**
//	**—**	**Null statement**
/*	**—**	**SYSIN data delimiter**

- **Note that there are other operations, some of which we will be discussing later in the class**

JCL Coding Rules

- **JCL is generally coded in uppercase**

  Exception: when coding parameters to access files in the z/OS UNIX File System (zFS), which is not covered in this course

- **Code up to column 72, then continue a statement, if necessary, on the next line as discussed on the next page...**

JCL Continuation

```
//NAME     OPERATION     OPERAND1,OPERAND2,
//        OPERAND3,OPERAND4,OPERAND5,
//        OPERAND6,     comments may go one or more spaces
//        OPERAND7,     after a comma
//        OPERAND8
```

- Continued statement must begin somewhere in columns 4-16, inclusive

JOB Statement Format

//jobname **JOB** **(***accounting info***),***progammer-name***,**

// **CLASS=***x***,MSGCLASS=***y***,**

// **NOTIFY=***userid***,TIME=(***min***,***sec***),**

// **TYPRUN={HOLD|SCAN}**

accounting info **- up to 143 characters, installation choice**

programmer name **- up to 20 characters, installation choice**

CLASS is job class, implying which initiators may run this job

- ✗ Installation runs some number of initiator address spaces for running batch jobs; each initiator is assigned to handle particular sets of job classes

MSGCLASS is SYSOUT class, specifying where printed / punched output from job should go

NOTIFY may specify a "*userid***" or "***node***.***userid***" ("node" option not supported in JES3)**

TIME special values: 1440 or NOLIMIT (both mean unlimited time), MAXIMUM (allows job to run up to 357,912 minutes - about 8 months)

Omit TYPRUN normally; SCAN checks for JCL errors, HOLD keeps initiator from selecting job until explicitly released

JOB Statement, continued

Examples

```
//MYWAY         JOB         (432,'RDD-343',NOXIOUS),JONES,
//      CLASS=A,MSGCLASS=H,NOTIFY=SJONES
.
.
.
```

```
//YOURWAY       JOB         (TR409,63),'O''NEIL',
//      NOTIFY=DEPT53.SSMITH,
//      TIME=2
.
.
.
```

- **Note: in JES3 systems, jobclass is specified on a JES3 MAIN statement, and job classes under JES3 can be up to 8 characters long**

Example:

```
//ANYWAY        JOB         (TRNG00P0),WIMP,
//      MSGCLASS=X,NOTIFY=WIMP
//*MAIN     CLASS=TSTHTEST
```

EXEC Statement Format

//stepname **EXEC** **{ PGM=***programname*
| PROC=*procedurename*
| *procedurename*}[,]

// **PARM=----------,**

// **TIME=(***min***,***sec***)**

❒ **You must specify one of PGM=, PROC=, or just a name (which is then assumed to be a "*procedurename*")**

"programname"

System will look for this name in the directory of the library of executable programs called SYS1.LINKLIB (or its extensions)

"procedurename"

System will look for this name in the directory of the library of pre-coded JCL called SYS1.PROCLIB (or its extensions)

❒ **PARM is any string up to 100 characters long to be passed directly to the program being run**

❒ **TIME has the same possibilities as for the JOB statement, plus you may code TIME=0 which means use any time remaining from the previous step**

EXEC Statement, continued

Examples

```
//STEP1          EXEC       PGM=ISDR01R
```

```
//STEP2          EXEC       PGM=XYZARG,
//      PARM='DEPARTMENT 56, SAN JOSE'
```

```
//STEPX          EXEC       PGM=SORT,TIME=(12,30)
```

```
//LOUSY          EXEC       PGM=BIGRPT,PARM='FINAL RE
//              PORT ON STUDIES DONE IN JANUARY'
```

Note continuation of quoted string

- ✗ string coded up to (and including) column 71
- ✗ continuation must have '//' in first two columns and continued text begins exactly in column 16

```
//CREDIT         EXEC       PROC=CPR43
```

```
//DISPUTE        EXEC       DSP567
```

Computer Exercise: OSWTO

Setup for all class labs:

Using ISPF option 6, enter the following command:

===> ex '.train.library(b610strt)' exec

(NOTE: you must code the fully qualified dataset name in quotes)

and press Enter.

This will cause the setup process to run. You will be prompted for a high level qualifier for your data sets. Unless the instructor tells you otherwise, use your TSO userid (the process is set up to use this as a default anyway). Press Enter.

The setup process will create a library for you to hold your JCL for the labs. The library name is <hlq>.TR.CNTL, where "<hlq>" is replaced by the high level qualifier you entered in response to the setup's prompt. This process also places a couple of members in your library you will need for various labs.

The lab ...

In your <hlq>.TR.CNTL data set, create a member called JCLEX01 to hold the Job Control statements necessary to run one job with two steps.

Reminder: to create a new member just use ISPF option 2 (edit); editing <userid>.TR.CNTL(JCLEX01) will create the member in your library, showing you an empty screen, ready to type.

First copy in member JOB at the front.

[From the command line enter: ===> *copy job*]

Next, since the program we are going to run is not stored in the standard system program library, we need to tell allocation where to find the program. After your JOB statement, and before any other JCL, code a statement like this:

```
//JOBLIB  DD  DISP=SHR,DSN=_____.TRAIN.LOADLIB
```

We'll discuss what this means later in the course.

Each of the two jobsteps should run the same program: OSWTO. So code two EXEC statements, each specifying PGM=OSWTO.

This program accepts from 1 to 25 characters (inclusive) from the PARM field on the EXEC statement and copies this data to the system message dataset (your JCL listing).

If you don't supply a value for the PARM field, the program will "blow up". On the other hand, if you supply more than 25 characters in the PARM field, the program will also "blow up".

So, on each step pass to the program, through the PARM field, your name (or userid) and the step name.

Do not run the job just yet - we have some more to talk about first.
(BUT... take a look at the next couple of pages for some ideas.)

This page intentionally left almost blank.

Clues for Writing JCL

- **Coding JCL is often a matter of "reading between the lines" of the information you are given, to translate this into JCL statements - looking for clues, as it were**
- **In the course of several exercises, we will sumarize pointers that are useful in most situations**

JOB Statements

Installation standards, and any JOB Statement generating edit macros normally provide you with all the information you need

```
//JOBname  JOB  (acctng),pgmr_name,CLASS=x,MSGCLASS=y[,
//       REGION=nn{K|M}[,TYPRUN={SCAN|HOLD}][,TIME=(min,sec)]  ]
```

- ✗ JOBname - see installation standards reference
- ✗ Accounting info - installation specific
- ✗ Programmer name - up to 20 characters; installation specific or programmer choice
- ✗ CLASS= - installation specific job class
- ✗ MSGCLASS= - installation specific SYSOUT class for JCL listings
- ✗ TYPRUN=SCAN - does a basic JCL syntax check, does not actually run the job
- ✗ TYPRUN=HOLD - holds job until explicitly released; used to run job in off shift, or to hold until another job has run
- ✗ REGION= - job dependent, if required; virtual storage necessary to run the job
- ✗ TIME= - minutes and seconds to allow the JOB to run before cancelling; use for testing, not production

Clues for Writing JCL, 2

EXEC Statements for Running Programs

Need an EXEC statement for each program to run in a job; the order of the EXEC statements is the order the programs will be run in

```
//stepname  EXEC  PGM=program_name[,PARM='  '][,TIME=][,
//      REGION=nn{K|M}]  ]
```

- ✗ Stepname - installation standard or programmer choice
- ✗ PGM= - name of program to run
- ✗ PARM= - up to 100 characters of parameter information; program specific; only code if you are told the program expects or needs certain PARM or parameter data, and you are also told what data to code
- ✗ TIME= - minutes / seconds this step is allowed to run; only code for testing, never production
- ✗ REGION= - program dependent, if required; virtual storage necessary to run the step

❒ **ALSO: you need to know where the programs are found**

If all programs are found in "the system libraries" or "the link list", then you do not need JOBLIB or STEPLIB statements

If a program is found in a particular library, you need to code

```
//STEPLIB  DD  DSN=library_name,DISP=SHR
```

or

```
//JOBLIB  DD  DSN=library_name,DISP=SHR
```

Place a STEPLIB after the EXEC statement it relates to; place a JOBLIB after the JOB statement