The Operating System Level

- Virtual Memory
- File systems
- Parallel processing
- Case studies

- Due 6/3: 2, 3, 18, 23

Like other levels we have studied, the OS level is built on top of the next lower layer.

Like other levels, the OS level exposes some of the lower level elements

e.g., the ISA level exposed registers, condition codes, memory (sometimes, somewhat).

The OS level exposes SOME of the ISA level. Often several ISA-level instructions are not available to the end user (with support from the OS level).

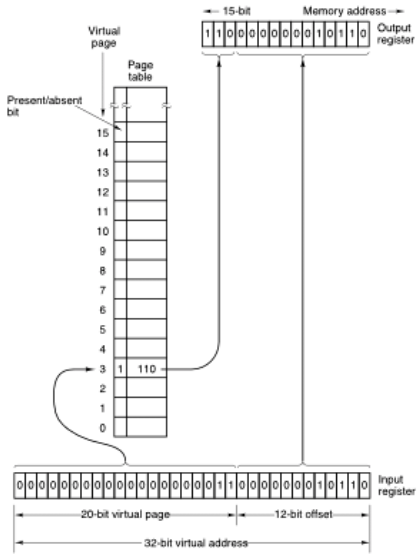
UNLIKE previous levels, the OS level is implemented completely in software (well, almost completely). That shouldn't obscure the fact that it is a genuine level, with its own vocabulary.

The assembly language programmer programs at the OS level: most of the ISA + a new set of "instructions" provided by the OS.

These are "system calls" - often implemented via a "trap" mechanism of the kind we discussed in the previous chapter.

Why? Because hardware support is needed to switch hardware modes and enable the additional, reserved, ISA instructions not available to normal programmers. These will typically involve the three areas above: virtual memory, file systems, and parallel processing.

Virtual memory



Page	Virtual addresses
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

Page frame	Bottom 32K of main memory Physical addresses
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

Physical memory vs addr space.

1. Just because a machine has a 32 bit addr space doesn't mean it actually has 4GB ram!

2. Even if it did, does that mean I can write STB 0x00, 0x00000000 (store a 0 in memory location zero)? What if the OS is in location 0?

Observation: most programs most of the time exhibit LOCALITY

Most of the memory references in an reasonably short period of time are near one another.

This is at a larger grain size that the locality exploited by cache, but still holds.

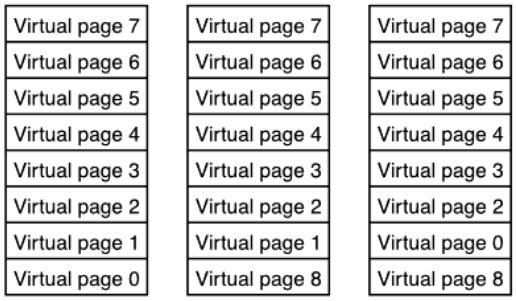
Solution: interpose a controlled mapping between ISA visible memory and physical memory. Note that, unlike cache, this is NOT another physical layer of memory!

However, like cache, it will be TRANSPARENT to the programmer.

Whereas the purpose of cache was to provide SPEED, the purpose of virtual memory is to provide SPACE - to provide the illusion of lots of memory.

Virtual memory

- Working Set



- Fragmentation

Ok, so we have a MECHANISM, now we need a POLICY for how to use it.

Demand Paging is a policy that says: when a page is referenced, get it

Huh? What else could you do? Well - think about instructions. Demand fetch would say get an instruction from memory when the PC gets there. Prefetch gets an instruction BEFORE it is needed. Generally don't do that in simplest version of demand paging - one reason it takes so long for a program to start up!

Ok, so that is policy for bringing pages in. Once we run out of room, we need a policy for getting rid of pages. Most common policy is LRU - least recently used. Another is FIFO. Issues are amount of hardware needed to keep track of information needed in policy and amount of time it takes to identify page to be removed, as well as effectiveness of method. E.g., LRU doesn't work well with loops of the wrong size. But then neither does FIFO.

Demand paging works if a program has a WORKING SET whose size fits in memory.

Consider: set of pages referenced in the first 10 ms of execution.

Now consider pages referenced in next 10 ms. If one new page in set, 50% cpu utilization for that time period. (because it will take 20 ms to execute: 10 to execute instr, and 10 to load missing page)

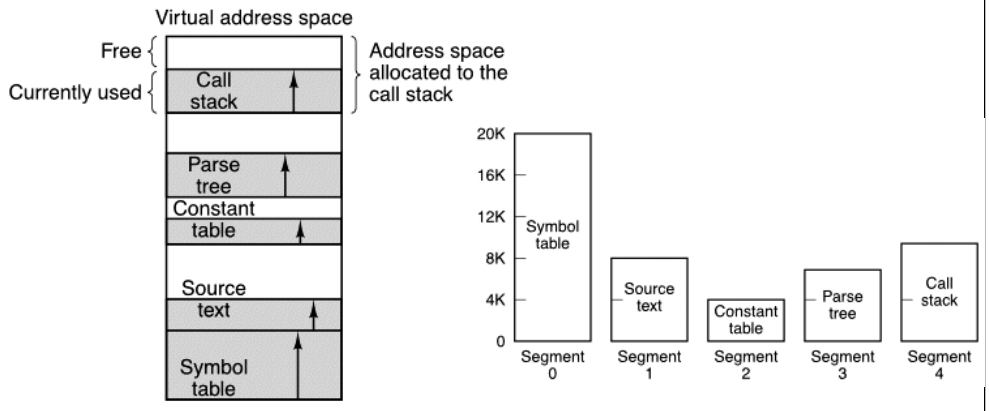
Assuming the set of pages referenced in the first 10 ms is much greater than 1, then we see that, for good cpu utilization, this set must be able to completely fit in memory, and must change slowly over time

Working set is the set of pages that reference one another (e.g., group of friends). When working set gets larger than available ram, THRASHING!

Comp question: 80%CPU, 96% disk utilization - thrashing? NO

20% CPU, 95% disk utilization - thrashing? YES

Segmentation



Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

So paging gives us a large linear address space.

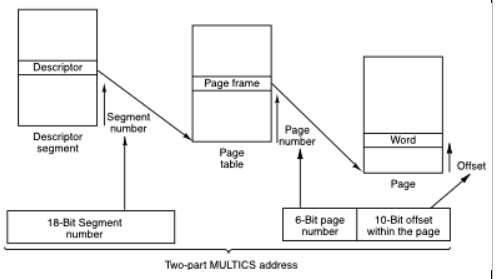
But sometimes it would be useful to have multiple addr spaces - e.g., multiple tables all growing dynamically.

SEGMENTATION does this:

a SEGMENT is a mapping of a chunk of address space from n,n+m to 0, 0+m

Note this is not the same as LV, CPP, etc. These are all sharing same physical address space.

Segmentation + VM?



Multics scheme (late 1960s)

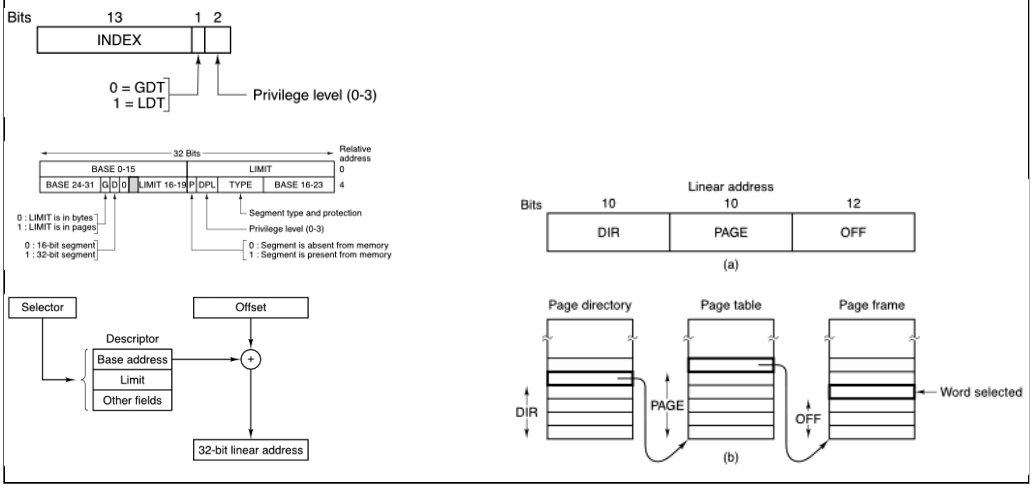
Every process had a descriptor segment.

The 18 bit segment number is an index into the process “Descriptor segment”, the corresponding entry contained the address of the page table for that segment.

Each 16 bit address then was a 6 bit page number followed by a 10 bit offset within page.

This scheme is close to that now used in the Pentium.

Pentium memory organization



Demand paging and segmentation

A “selector” is in CS or DS (code selector or data selector) register.

GDT - global (ie, OS) selector

LDT - local (ie, this process) selector.

Talk about this - draw a picture of local and global memory segements.

Segment descriptors are 64 bits long - handy - that means putting the GDT or LDT index into the top 13 bits we can just add three zeros to convert an index into an offset.

Segment descriptor is somewhat complicated, all we really care about is that it has a base and a limit.

Next figure shows how base gets added to offset in instruction

Like: addc (R2, 3), R4 - (R2, 3) means take value of R2, add three,

that is the OFFSET from the start of the data sector.

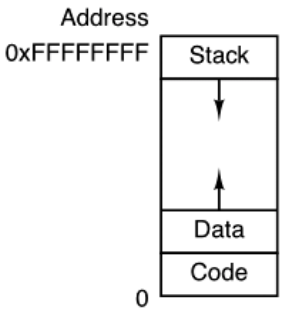
This, finally, results in an address into the 32 bit virtual address space.

4 k byte pages mean potentially 1 million pages!

We don’t want to actually make room for a 1 million entry page table!

So, a directory of actual page tables, the directory holds 1000 entries, each to a page table with addresses of 1000 pages. If a page table is empty, its directory entry says “invalid range” and we don’t need to allocate memory for that page table.

OS use of VM



Every process has its own virtual address space, 32 bits long

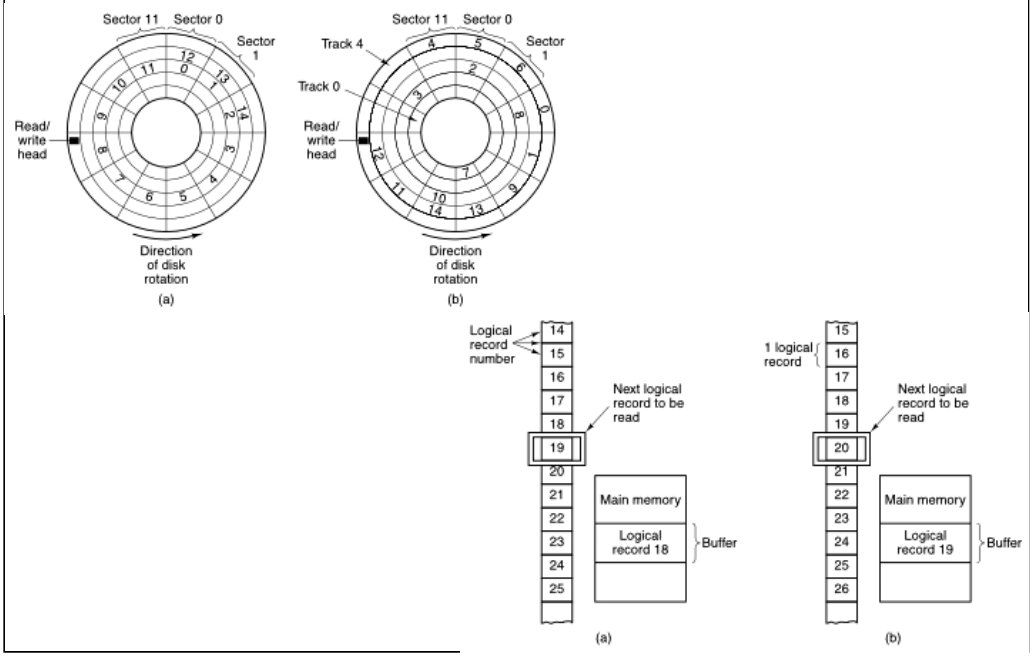
Lower 2GB are process, upper 2GB are mapped to kernel.

Each committed page (page with data or code in it) has a “shadow page” on disk

Unix:

Three segments: code, data, stack.

File Systems.



The first OS-level addition, virtual memory, is largely invisible to the programmer.

The second, “virtual I/O”, is not.

I/O at the ISA level is quite obscure and device dependent. A particular device might require a particular sequence of operations, expose a particular set of status bits, have unique timing and data rate (and format) requirements, etc.

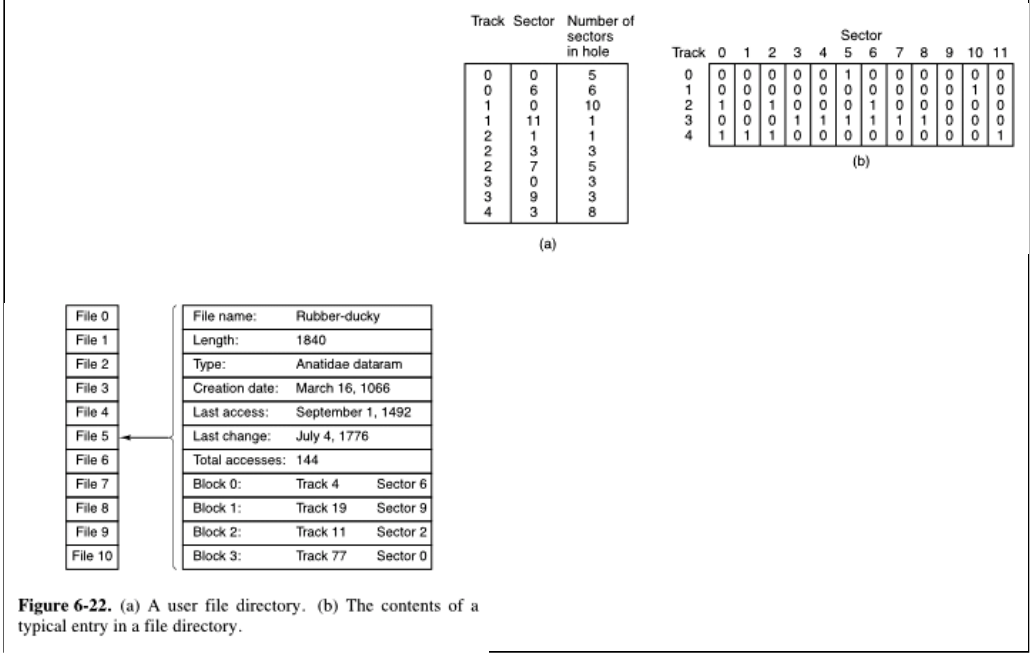
Much of this is masked at the OS-level, using an abstraction called the “file”.

Common abstraction: sequence of bytes.

Alternate abstraction: sequence of *records*

Why ***sequence***? Because of physical character istics of disk: sequential access is much more efficient than random access.

Space management



Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

(a)

Track	Sector 0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

File 0	File name:	Rubber-ducky
File 1	Length:	1840
File 2	Type:	Anatidae dataram
File 3	Creation date:	March 16, 1066
File 4	Last access:	September 1, 1492
File 5	Last change:	July 4, 1776
File 6	Total accesses:	144
File 7	Block 0:	Track 4 Sector 6
File 8	Block 1:	Track 19 Sector 9
File 9	Block 2:	Track 11 Sector 2
File 10	Block 3:	Track 77 Sector 0

Figure 6-22. (a) A user file directory. (b) The contents of a typical entry in a file directory.

Two key issues: how to keep track of space, how to keep track of files.

Remember the problem that lead us to introduce memory management? Too many tables each growing unpredictably, running into each other.

Solution was virtual memory. Same solution occurs on disk: when we need a new spot on disk to store some data, where do we get it?

Difference is “page” (allocation unit) size is often not fixed: if we know we are going to need 10 mb, better to allocate it up front.

A **FILE INDEX** maps from sequential bytes in a file to allocation units on disk.

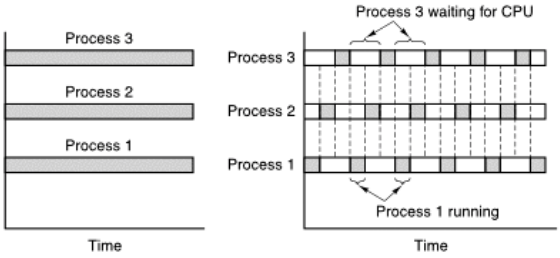
Hole list vs bit map.

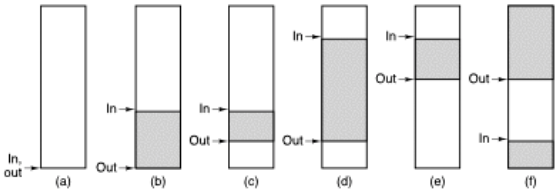
Why is this important? Mostly just to expose you to a grab-bag of solutions.

Directories help users (and the OS) find files, just like your phone book helps you find someone’s phone number or address.

Parallel Processing

- Process
  - Creation
  - Suspension
  - Inspection
  - Resumption
  - Termination
  - Synchronization





Parallel processing, whether real or simulated, is increasingly important.

I watch a dvd while I work on slides for class and my email program checks for new mail.

2-4 processors Intel and Apple machines are common.

What's the difference between a program and a process?

In general, a program is something the user starts (or a programmer writes), whereas a process is a separate program counter. A program can have many processes, and an OS will often have many processes active even when no programs (user) are runnning.

Simulation of multiprocessing on a single processor.

OS-level instructions are provided for working with processes. These vary by OS, and we won't examine in any detail except for synchronization. Let's look at that.

In particular, let's start by looking at a java solution to the ciruclar buffer.

Java for circular buffer

```
class producer extends Thread {          // producer class
   public void run( ) {                  // producer code
      long prime = 2;                    // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);      // statement P1
         if (m.next(m.in) == m.out) suspend( );      // statement P2
         m.buffer[m.in] = prime;         // statement P3
         m.in = m.next(m.in);            // statement P4
         if (m.next(m.out) == m.in) m.c.resume( );  // statement P5
      }
   }
```

IN

OUT

```
class consumer extends Thread {          // consumer class
   public void run( ) {                  // consumer code
      long emirp = 2;                    // scratch variable

      while (emirp < m.MAX_PRIME) {
         if (m.in == m.out) suspend( );  // statement C1
         emirp = m.buffer[m.out];        // statement C2
         m.out = m.next(m.out);          // statement C3
         if (m.out == m.next(m.next(m.in))) m.p.resume( );//
         System.out.println(emirp);      // statement C5
      }
   }
```

```
public class m {
   final public static int BUF_SIZE = 100; // buffer runs from 0 to 99
   final public static long MAX_PRIME = 100000000000L; // stop here
   public static int in = 0, out = 0;          // pointers to the data
   public static long buffer[ ] = new long[BUF_SIZE];// primes stored here
   public static producer p;                   // name of the producer
   public static consumer c;                   // name of the consumer

   public static void main(String args[ ]) { // main class
      p = new producer( );                     // create the producer
      c = new consumer( );                     // create the consumer
      p.start( );                              // start the producer
      c.start( );                              // start the consumer
   }

   // This is a utility function for circularly incrementing in and out
   public static int next(int k) {if (k < BUF_SIZE – 1) return(k+1); else return(0);}
}
```

Ok, let's see how this works.

We have two pointers into the circular buffer, in and out. In points to the next empty spot, out points to the next spot with a prime to be taken.

Producer is producing a series or primes. Each time it makes on, it checks if there is room in the circular buffer, and if so, adds it to the buffer, incrementing the pointer to the next empty spot

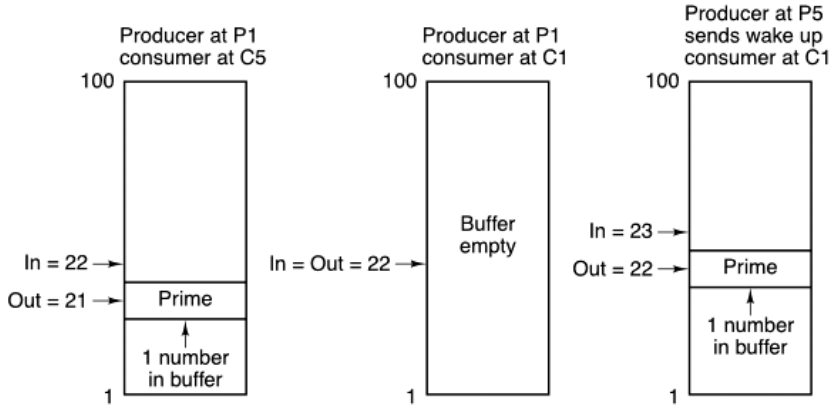
If not, producer *suspends*.

Consumer, similarly, checks to see if there is an available prime (in != out), and if so gets it and increments out. Otherwise, it suspends.

Note each also checks if it needs to wake up the other: consumer wakes producer if it has now created one empty spot (ie, if buffer was full prior to it removing prime) and vice-versa.

So, looks great, what could be the problem?

Circular buffer problem



Consider when only one number in buffer.

IN = 22, OUT = 21

Producer at P1, consumer at C5.

Consumer gets next prime, and increments out. Now IN==OUT==22

Consumer then goes to C1 and gets in and out to compare.

RIGHT THEN, after fetch but before compare

Producer finds next prime, puts it in buffer, and increments IN. (P3 & P4)

At P5, it decides that, since IN = next (out), Consumer must be suspended, so it *resume*s Consumer.

But, consumer hasn't suspended yet!

Now consumer starts up again, and finds its copies of IN ==OUT, so it suspends.

Producer keeps producing primes untill buffer fills, then it suspends.

And you sit there wondering what you did wrong. (worse, sometimes the rpoigram runs, other times it dies at random locations in the sequence of primes, and for the life of you you can't figure out why!

This is called a *race condition*.

Semaphores

```
class producer extends Thread {          // producer class
   native void up(int s); native void down(int s);    // methoc
   public void run( ) {                  // producer code
      long prime = 2;                    // scratch variable

      while (prime < m.MAX_PRIME) {
         prime = next_prime(prime);      // statement P1
         down(m.available);              // statement P2
         m.buffer[m.in] = prime;         // statement P3
         m.in = m.next(m.in);            // statement P4
         up(m.filled);                   // statement P5
      }
   }

   private long next_prime(long prime){ ... } // function that
}
 class consumer extends Thread {         // consumer class
    native void up(int s); native void down(int s);    // method
    public void run( ) {                 // consumer code
       long emirp = 2;                   // scratch variable

       while (emirp < m.MAX_PRIME) {
          down(m.filled);                // statement C1
          emirp = m.buffer[m.out];       // statement C2
          m.out = m.next(m.out);         // statement C3
          up(m.available);               // statement C4
          System.out.println(emirp);     // statement C5
       }
    }
```

Semaphores are a cleanly thought out solution to this problem.

A Semaphore is an object that supports two operations: *up* and *down*

Up: semaphore = semaphore +1

if semaphore was 0 and other process was halted attempting a down, allow it to continue and complete the down

Down: if semaphore = 0, suspend;

semaphore --;

This works as long as up and down are indivisible operations.

On single processor machines, this can be done by supressing interupts. On multi-processors, there is often an hardware instruction of the form: decrement, store, and set condition-codes.

(what about piplelined and superscalar architectures? They must make such instructions appear indivisible.)

If an interrupt does occur, condition-codes are usually stored.