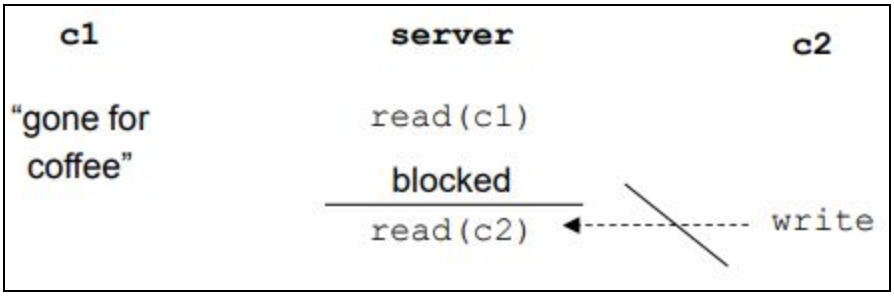
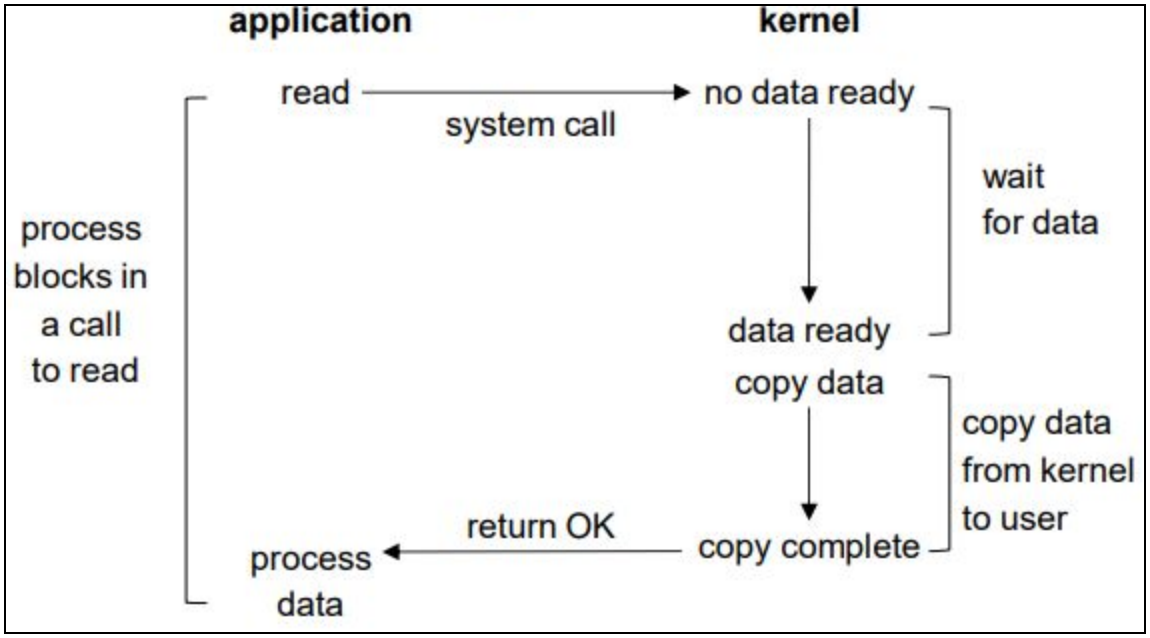
Week 11 Notes

1. **I/O Multiplexing:**
   - **I/O multiplexing** is a process which is used to monitor multiple file descriptors and choose among them which fd is ready for reading or writing to a socket.
   - Consider this problem: A server is ready to read with its clients. However, the server can only read from 1 client at a time. If the first client is gone, then all the other clients are blocked.
     I.e.

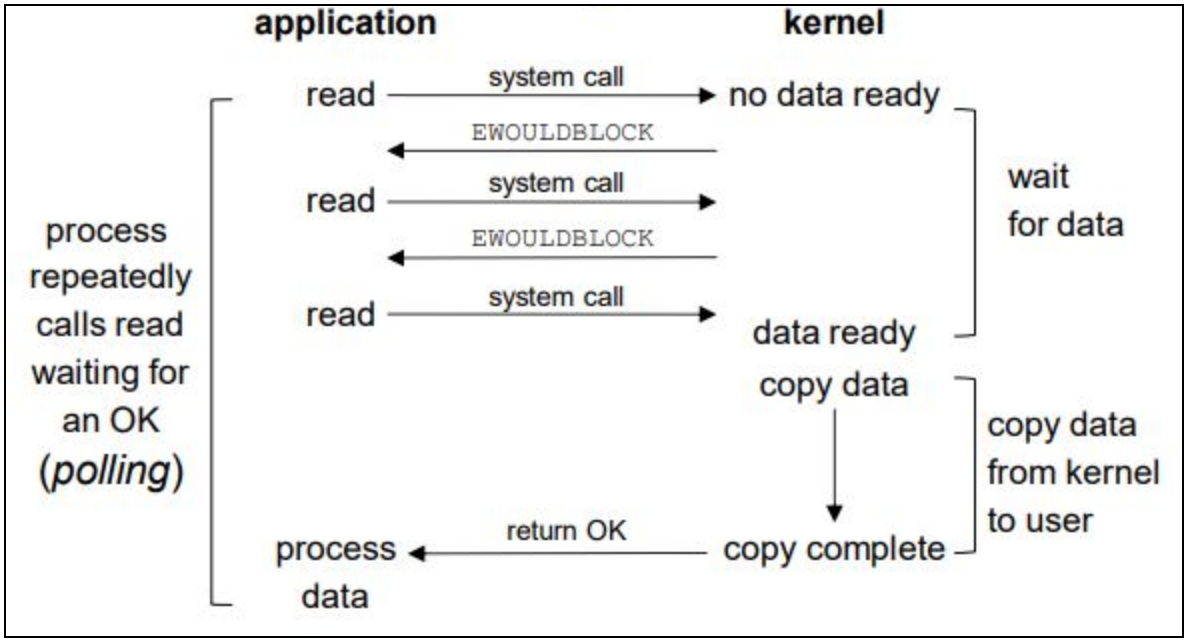


   - **Note:** If you are using sockets, read and accept will both block until the client is done.
   - When reading from multiple sources, blocking on one of the sources could be bad. A possible and simple solution would be to create one process for every client. However, a downside to this is that if there a lot of clients, then a lot of processes have to be made, which is inefficient. I/O multiplexing can solve this problem.
   - There are normally two distinct phases for an input operation:
     1. Waiting for the data to be ready. This involves waiting for data to arrive on the network. When the packet arrives, it is copied into a buffer within the kernel.
     2. Copying the data from the kernel to the process. This means copying the (ready) data from the kernel's buffer into our application buffer.

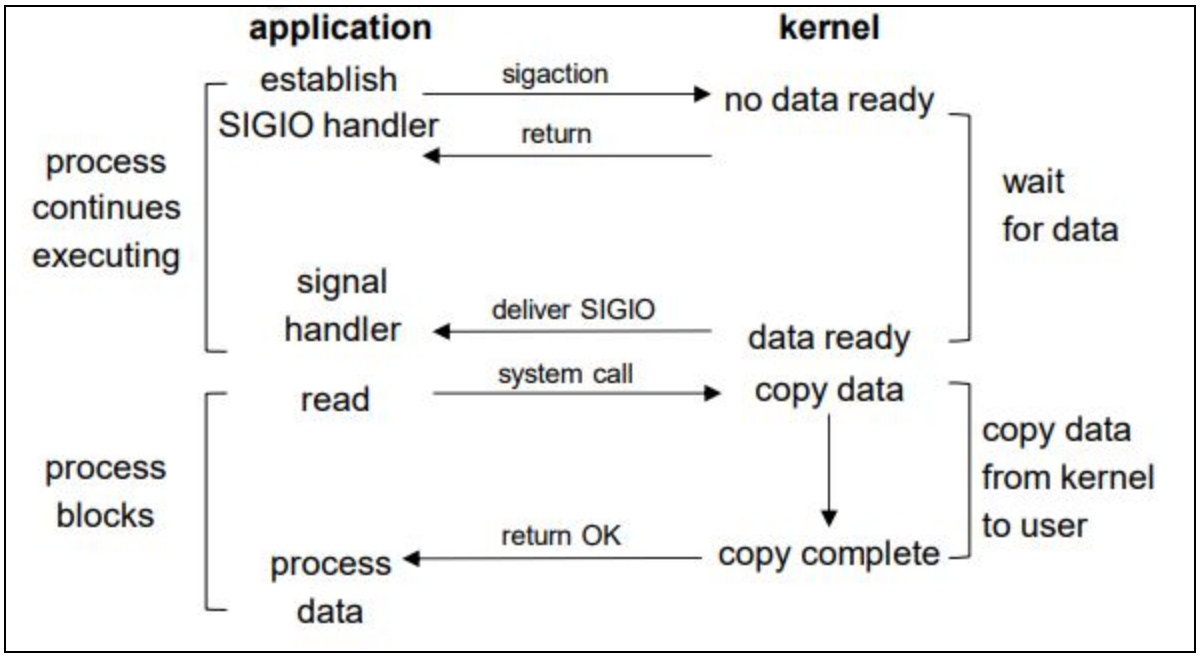
2. **Blocking I/O Model:**
   - The most prevalent model for I/O is the blocking I/O model.
   - By default, all sockets are blocking.



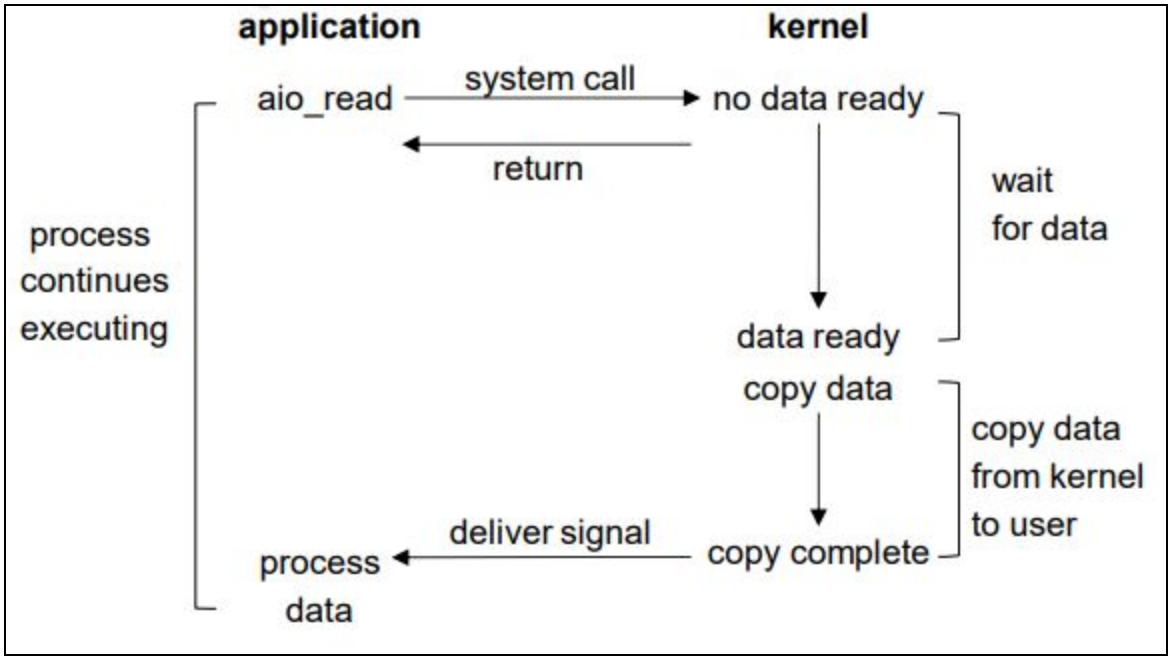
3. **Nonblocking I/O Model:**
   - When a socket is set to be non-blocking, we are telling the kernel "When an I/O operation that I requested cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead".



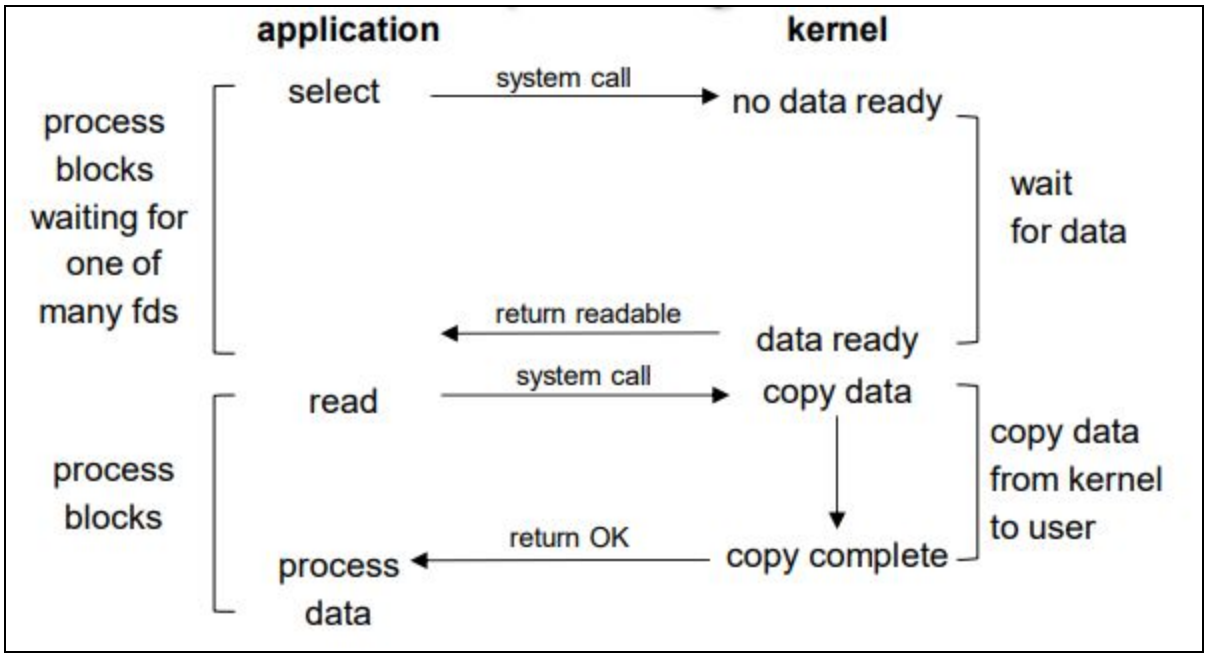
4. **Signal Driven I/O Model:**
    - The signal driven I/O Model tells the kernel to notify us with the SIGIO signal when the descriptor is ready.
    - We first enable the socket for signal-driven I/O and install a signal handler using the sigaction system call. The return from this system call is immediate and our process continues; it is not blocked.
    - When the datagram is ready to be read, the SIGIO signal is generated for our process.
    - The advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.



5. **Asynchronous I/O Model:**
    - This works by telling the kernel to start the operation and to notify us when the entire operation,including the copy of the data from the kernel to our buffer, is complete.
    - The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete.
    - We call aio_read and pass the kernel the following:
        - A file descriptor, file offset, buffer pointer and buffer size.
        - How to notify us when the entire operation is complete.
    - This system call returns immediately and our process is not blocked while waiting for the I/O to complete.


application
kernel
aio_read
system call
no data ready
return
wait
for data
process
continues
executing
data ready
copy data
copy data
from kernel
to user
deliver signal
process
data
copy complete

6. **I/O Multiplexing Model:**
   - With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call.
   - We block in a call to select, waiting for the datagram socket to be readable. When select returns that the socket is readable, we then call read to copy the datagram into our application buffer.
   - Disadvantage: Using select requires two system calls, select and read, instead of one.
   - Advantage: We can wait for more than one file descriptor to be ready.



7. **Select:**
   - Select allows you to monitor several FDs without using fork or tight looping.
   - Select allows a process to instruct the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait.
   - Sometimes a program needs to accept input on multiple input channels whenever input arrives. E.g. A program that acts as a server to several other processes via pipes or sockets. You cannot normally use read for this purpose, because read blocks the program until input is available on one particular file descriptor. A solution is to use select. Select blocks the program until input or output is ready on a specified set of file descriptors, or until a timer expires, whichever comes first.
   - Select blocks until at least one of these FDs has an action.
   - The FDs that remain in the FD sets are the ones on which one can read and write without blocking.
   - Syntax: **int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);**

- The fd_set data type represents file descriptor sets for the select function. It is actually a bit array.
- maxfdp1 specifies the number of descriptors to test. Its value is the maximum descriptor to be tested plus one.
- The reason the maxfdp1 argument exists, along with the burden of calculating its value, is for efficiency. Although each fd_set has room for many descriptors, typically 1024, this is much more than the number used by a typical process. The kernel gains efficiency by not copying unneeded portions of the descriptor set between the process and the kernel, and by not testing bits that are always 0.
- A call to select returns when one of the file descriptors in one of the sets is ready for I/O.
- Select returns the positive number of ready descriptors on success, 0 if we timed out, and -1 on error.
- Select is ready to read when:
    - There is data in the receive buffer to be read.
    - End-of-file state on file descriptor.
    - The socket is a listening socket and there is a connection pending.
    - A socket error is pending and you need to catch it.
- Select is ready to write when:
    - There is space available in the write buffer.
    - A socket error is pending and you need to catch it.
- Select is ready to handle exception conditions when:
    - There is TCP out-of-band data.
- Typically, only the readset is used.
- Timeout specifies how long we're willing to wait for a fd to become ready.
- **struct timeval {**
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds */
  };
- There are 3 cases for timeout:
    - If timeout is NULL, we wait forever or until we catch a signal.
    - If timeout is zero, we test and return immediately.
    - Otherwise, we wait up to specified time.

8. **Descriptor Sets:**
    - Typically implemented as an array of integers where each bit corresponds to a descriptor (except in Windows).
    - Implementation is hidden in the fd_set data type.
    - FD_SETSIZE is the number of descriptors in the data type.
    - Macros:
        - **void FD_ZERO(fd_set *fdset);** //Clears all bits in fdset.
        - **void FD_SET(int fd, fd_set *fdset);** //Turn on the bit for fd in fdset.
        - **void FD_CLR(int fd, fd_set *fdset);** //Turn off the bit for fd in fdset.
        - **int FD_ISSET(int fd, fd_set *fdset);** //Is the bit for fd on in fdset?

- E.g. Consider the code snippet below.

```
fd_set rset;

FD_ZERO(&rset);          // Initialize the set. All bits are off.
FD_SET(1, &rset);        // Turn on bit for fd 1.
FD_SET(4, &rset);        // Turn on bit for fd 4.
FD_SET(5, &rset);        // Turn on bit for fd 5.
```

- Select modifies the descriptor sets pointed to by the readset, writeset, and exceptset pointers.When we call the function, we specify the values of the descriptors that we are interested in, and on return, the result indicates which descriptors are ready. We use the FD_ISSET macro on return to test a specific descriptor in an fd_set structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set. To handle this, we turn on all the bits in which we are interested in all the descriptor sets each time we call select.

9. **Bit Strings:**
   - Signal mask and file descriptor sets are implemented using bit array or bit strings.
   - Each bit represents an element of the set:
     - 1 means that the element is in the set.
     - 0 means that the element is not in the set
   - **Bitwise operators:**

Operator and Name	Description
& (AND)	It copies a bit to the result if it exists in both operands.
| (OR)	It copies a bit to the result if it exists in either operand.
^ (XOR)	It copies the bit to the result if it is in exactly one operand.
~ (Complement)	Flips all 0's to 1's and vice versa.
<< (Left shift)	All bits are shifted left by the given number.
>> (Right shift)	All bits are shifted left by the given number.

   - **Note:** Left and right shift will cause bits to fall off the ends.
   - E.g. Suppose that A is 0011 1100 and B is 0000 1101. Then:
     - (A&B) is 0000 1100
     - (A|B) is 0011 1101
     - (A^B) is 0011 0001
     - (~A) is 1100 0011
     - (A << 2) is 1111 0000
     - (B >> 3) is 0000 0001

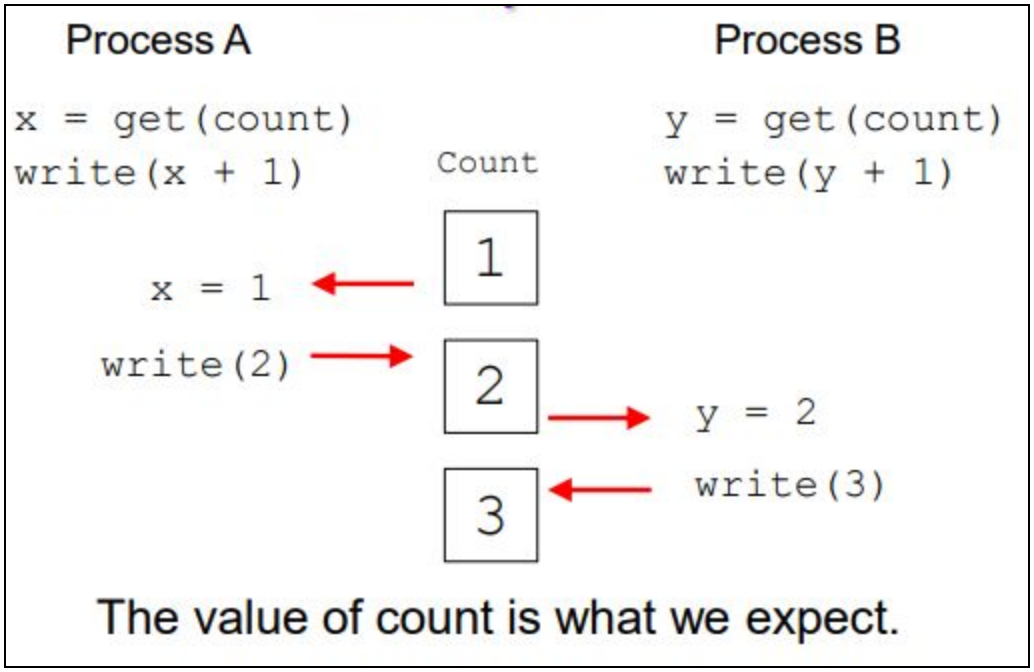
Week 11 Notes

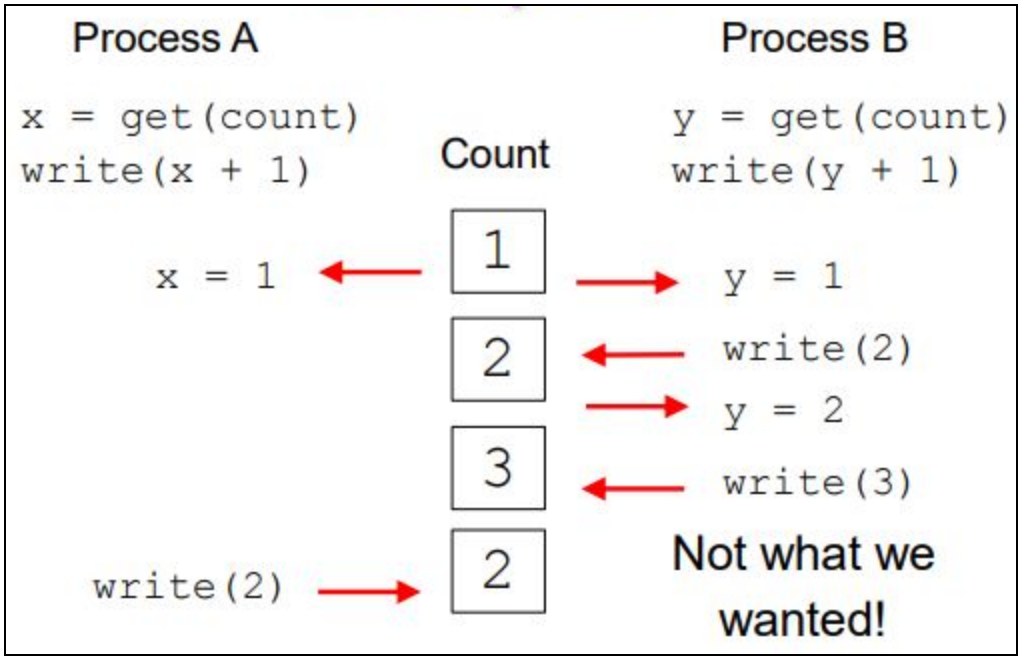
- Precedence of bit operators:
  Highest: ~
  &
  ^
  Lowest: |
- Flags are often implemented as a bit mask.
  E.g.
  open("temp", O_WRONLY | O_CREAT);
  #define O_RDONLY 00
  #define O_WRONLY 01
  #define O_RDWR 02
  #define O_CREAT 0100
- FD_SETSIZE is bigger than 32.
- **struct bits {**
  unsigned int field[N];
  }
  typedef struct bits Bitstring;
  Bitstring a, b;
  setzero(&a);
  b = a;
  a.field[0] = ~0;
- The set function sets one bit to 1. This is predefined in the C library.
  int set(unsigned int bit, Bitstring *b) {
  int index = bit / 32;
  b->field[index] |= 1 << (bit % 32);
  return 1;
  }
- The unset function sets one bit to 0. This is predefined in the C library.
  int unset(unsigned int bit, Bitstring *b) {
  int index = bit / 32;
  b->field[index] &= ~(1 << (bit % 32));
  }
- The ifset function checks if a bit is set in the bit string. This is predefined in the C library.
  int ifset(unsigned int bit, Bitstring *b) {
  int index = bit / 32;
  return ((1 << (bit % 32)) & b->field[index]);
  }
- The setzero function sets all bits to 0. This is predefined in the C library.
  int setzero(Bitstring *b){
  if(memset(b,0, sizeof(Bitstring)) == NULL)
  return 0;
  else
  return 1;
  }

- The intToBinary function converts an int to binary. This is predefined in the C library.

```
char *intToBinary(unsigned int number) {
        char *binaryString = malloc(32+1);
        int i; binaryString[32] = '\0';
        for (i = 31; i >= 0; i--) {
                binaryString[i] = ((number & 1) + '0');
                number = number >> 1;
        }
        return binaryString;
}
```

10. **Concurrency:**
    - The two key concepts driving computer systems and applications are:
        - **Communication:** The conveying of information from one entity to another.
        - **Concurrency:** The sharing of resources in the same time frame.
    - Concurrency can exist in a single processor as well as in a multiprocessor system.
    - Managing concurrency is difficult, as execution behaviour is not always reproducible.
11. **Race conditions:**
    - A **race condition** occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
    - E.g.





12. **Producer/Consumer Problem:**
    - The scenario goes as follows:
        - Suppose there are two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming (removing) the data, one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
    - The consumer should be blocked when buffer is empty.
    - The producer should be blocked when buffer is full.
    - The producer and consumer should run independently as far as buffer capacity and contents permit.
    - The producer and consumer should never be updating the buffer at the same time (otherwise data integrity cannot be guaranteed).
    - The producer/consumer is a harder problem if there are more than one consumer and/or more than one producer.
    - Programs that manage shared resources must protect the integrity of the shared resources.
    - Operations that modify the shared resource are called critical sections.
    - Critical section must be executed in a mutually exclusive manner.
    - Semaphores are commonly used to protect critical sections.

13. <u>Semaphores:</u>

- Code that modifies shared data usually has the following parts:
  - **Entry section:** The code that requests permission to modify the shared data.
  - **Critical Section:** The code that modifies the shared variable.
  - **Exit Section:** The code that releases access to the shared data.
  - **Remainder:** The remaining code.
- Useful in process synchronization and multithreading.
- The way semaphores work is that:
  1. You call a function, acquire.
     acquire(v): Blocks until the value of the semaphore variable v until it is greater than 0 then it decrements v.
  2. You call a function, release.
     release(v): Increments v.
- The structure of how semaphores work is that:
  1. Create a semaphore variable v.
  2. init(v)
  3. acquire(v)
  4. Critical section of code.
  5. release(v)
  6. Remainder of code.