It's not unusual to find that an application is organized as multiple communicating processes.

What's the advantage of using multiple processes instead of just a single process?

Many applications exhibit concurrency, i.e., some of the required computations can be performed in parallel.

For example, video compression algorithms represent each video frame as an array of 8-pixel by 8-pixel macroblocks.

Each macroblock is individually compressed by converting the 64 intensity and color values from the spatial domain to the frequency domain and then quantizing and Huffman encoding the frequency coefficients.

If you're using a multi-core processor to do the compression, you can perform the macroblock compressions concurrently.

Applications like video games are naturally divided into the "front-end" user interface and "back-end" simulation and rendering engines.

Inputs from the user arrive asynchronously with respect to the simulation and it's easiest to organize the processing of user events separately from the backend processing.

Processes are an effective way to encapsulate the state and computation for what are logically independent components of an application, which communicate with one another when they need to share information.

These sorts of applications are often data- or event-driven, i.e., the processing required is determined by the data to be processed or the arrival of external events.

How should the processes communicate with each other?

If the processes are running out of the same physical memory, it would be easy to arrange to share memory data by mapping the same physical page into the contexts for both processes.

Any data written to that page by one process will be able to be read by the other process.

To make it easier to coordinate the processes' communicating via shared memory, we'll see it's convenient to provide synchronization primitives.

Some ISAs include instructions that make it easy to do the required synchronization.

Another approach is to add OS supervisor calls to pass messages from one process to another.

Message passing involves more overhead than shared memory, but makes the application programming independent of whether the communicating processes are running on the same physical processor.

In this lecture, we'll use the classic producer-consumer problem as our example of concurrent processes that need to communicate and synchronize.

There are two processes: a producer and a consumer.

The producer is running in a loop, which performs some computation to generate information, in this case, a single character C. The consumer is also running a loop, which waits for the next character to arrive from the producer, then performs some computation .

The information passing between the producer and consumer could obviously be much more complicated than a single character.

For example, a compiler might produce a sequence of assembly language statements that are passed to the assembler to be converted into the appropriate binary representation.

The user interface front-end for a video game might pass a sequence of player actions to the simulation and rendering back-end.

In fact, the notion of hooking multiple processes together in a processing pipeline is so useful that the Unix and Linux operating systems provide a PIPE primitive in the operating system that connects the output channel of the upstream process to the input channel of the downstream process.

Let's look at a timing diagram for the actions of our simple producer/consumer example.

We'll use arrows to indicate when one action happens before another.

Inside a single process, e.g., the producer, the order of execution implies a particular ordering in time: the first execution of is followed by the sending of the first character.

Then there's the second execution of , followed by the sending of the second character, and so on.

In later examples, we'll omit the timing arrows between successive statements in the same program.

We see a similar order of execution in the consumer: the first character is received, then the computation is performed for the first time, etc.

Inside of each process, the process' program counter is determining the order in which the computations are performed.

So far, so good - each process is running as expected.

However, for the producer/consumer system to function correctly as a whole, we'll need to introduce some additional constraints on the order of execution.

These are called "precedence constraints" and we'll use this stylized less-than sign to indicate that computation A must precede, i.e., come before, computation B.

In the producer/consumer system we can't consume data before it's been produced, a constraint we can formalize as requiring that the i_th send operation has to precede the i_th receive operation.

This timing constraint is shown as the solid red arrow in the timing diagram.

Assuming we're using, say, a shared memory location to hold the character being transmitted from the producer to the consumer, we need to ensure that the producer doesn't overwrite the previous character before it's been read by the consumer.

In other words, we require the i_th receive to precede the i+1_st send.

These timing constraints are shown as the dotted red arrows in the timing diagram.

Together these precedence constraints mean that the producer and consumer are tightly coupled in the sense that a character has to be read by the consumer before the next character can be sent by the producer, which might be less than optimal if the and computations take a variable amount of time.

So let's see how we can relax the constraints to allow for more independence between the producer and consumer.

We can relax the execution constraints on the producer and consumer by having them communicate via N-character first-in-first-out (FIFO) buffer.

As the producer produces characters it inserts them into the buffer.

The consumer reads characters from the buffer in the same order as they were produced.

The buffer can hold between 0 and N characters.

If the buffer holds 0 characters, it's empty; if it holds N characters, it's full.

The producer should wait if the buffer is full, the consumer should wait if the buffer is empty.

Using the N-character FIFO buffer relaxes our second overwrite constraint to the requirement that the i_th receive must happen before i+N_th send.

In other words, the producer can get up to N characters ahead of the consumer.

FIFO buffers are implemented as an N-element character array with two indices: the read index indicates the next character to be read, the write index indicates the next character to be written.

We'll also need a counter to keep track of the number of characters held by the buffer, but that's been omitted from this diagram.

The indices are incremented modulo N, i.e., the next element to be accessed after the N-1_st element is the 0_th element, hence the name "circular buffer".

Here's how it works.

The producer runs, using the write index to add the first character to the buffer.

The producer can produce additional characters, but must wait once the buffer is full.

The consumer can receive a character anytime the buffer is not empty, using the read index to keep track of the next character to be read.

Execution of the producer and consumer can proceed in any order so long as the producer doesn't write into a full buffer and the consumer doesn't read from an empty buffer.

Here's what the code for the producer and consumer might look like.

The array and indices for the circular buffer live in shared memory where they can be accessed by both processes.

The SEND routine in the producer uses the write index IN to keep track of where to write the next character.

Similarly the RCV routine in the consumer uses the read index OUT to keep track of the next character to be read.

After each use, each index is incremented modulo N.

The problem with this code is that, as currently written, neither of the two precedence constraints is enforced.

The consumer can read from an empty buffer and the producer can overwrite entries when the buffer is full.

We'll need to modify this code to enforce the constraints and for that we'll introduce a new programming construct that we'll use to provide the appropriate inter-process synchronization.