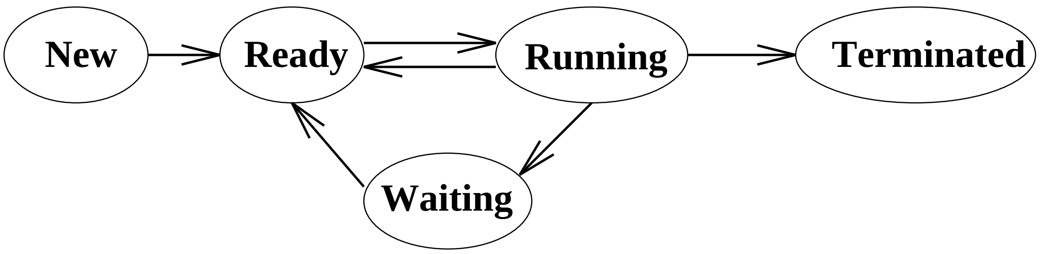
- Each process maintains its own state, which includes its text and data, procedure call stack, heap space, etc.
- The OS also stores process state for each process. This state is called the *Process Control Block* (PCB), and it includes the PC, SP, other register contents, execution state, etc.
- Process Execution State



- For each of these execution states, the OS maintains a state queue. Each process that the OS is currently managing resides in exactly one of these state queues.

- How do processes communicate? Is this efficient?
- What is the difference between a process and a thread?
- What are the advantages and disadvantages of using threads versus processes?

Cooperating Processes

Processes can cooperate with each other to accomplish a single task. Cooperating processes can

- improve performance by overlapping activities or performing work in parallel,
- enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program, and
- easily share information between tasks.

$\Longrightarrow$ Distributed and parallel processing is the wave of the future. To program these machines, we must cooperate and coordinate between separate processes.

Cooperating Processes

Example: A common use of cooperating processes is a producer process and one or more consumer processes.

```
n = 100;    // max outstanding items
buffer [n];
in = 0;
out = 0;
producer
    repeat forever {
        . . .
        nextp = produced item
        . . .
        // Make sure buffer is not full
        while in+1 mod n = out do
            no-op
        buffer[in] = nextp
        in = in+1 mod n
    }
```

```
consumer
    repeat forever {
        // Make sure buffer
        // is not empty
        while in = out do no-op
        nextc = buffer[out]
        out = out+1 mod n
        . . .
        consume nextc
        . . .
    }
```

What would happen if we wrote a producer program that forked a process to run the consumer in?

How Cooperating Processes Communicate

1. Communicating with message passing (distributed systems use this solution).

```
main()
    . . .
    if (fork() != 0) producerSR();
    else consumerSR();
end
```

```
producerSR                              consumerSR
    repeat                                  repeat
        . . .                                   receive (nextc, producer)
        produce item nextp                      . . .
        . . .                                   consume item nextc
        send (nextp, consumer)                  . . .
```

- Each process needs to be able to name the other process.
- The consumer is assumed to have an infinite buffer size.
- A bounded buffer would require the tests in the previous slide, and communication of the `in` and `out` variables (`in` from producer to consumer, `out` from consumer to producer).
- OS keeps track of messages (copies them, notifies receiving process, etc.).

⟹ How would you use message passing to implement a single producer and multiple consumers?

How Cooperating Processes Communicate

2. Communicating through shared memory:
   - establish a mapping between the process's address space to a named memory object that may be shared across processes. (Under SunOS, the system call mmap(. . .) performs this function.)
   - Fork processes that need to share the data structure.

```
main()
    . . .
    mmap(. . ., in, out, PROT_WRITE, MAP_SHARED, . . .);
    in = 0;
    out = 0;
    if (fork() != 0) producer();
    else consumer();
end
```

```
producer
repeat
    . . .
    produce item nextp
    . . .
    while in+1 mod n = out do no-op
    buffer[in] = nextp
    in = in+1 mod n
```

```
consumer
    repeat
    while in = out do no-op
    nextc = buffer[out]
    out = out+1 mod n
    . . .
    consume item nextc
    . . .
```

Scheduling and Overhead of Cooperating Processes

The OS decides policy - which process to run and when to run it.

Problem: In the shared-memory producer/consumer example, the user would only want to run the consumer if the producer had actually produced something. The user would not want the producer to run if the buffer was full. If the buffer was empty, the user would not want the consumer to run.

Bottom line:

1. OS processes do not give users any control over process schedules.
2. Even though the processes share data and are cooperating, switching between them has the same (relatively expensive) cost as every other context switch.

$\Longrightarrow$ These drawbacks lead to a refined definition of process that separates the *thread* of control from the address space and other attributes of the process.

Processes versus Threads

Threads extract the *thread of control* information from the process, which enables multiple threads to operate within the same process.

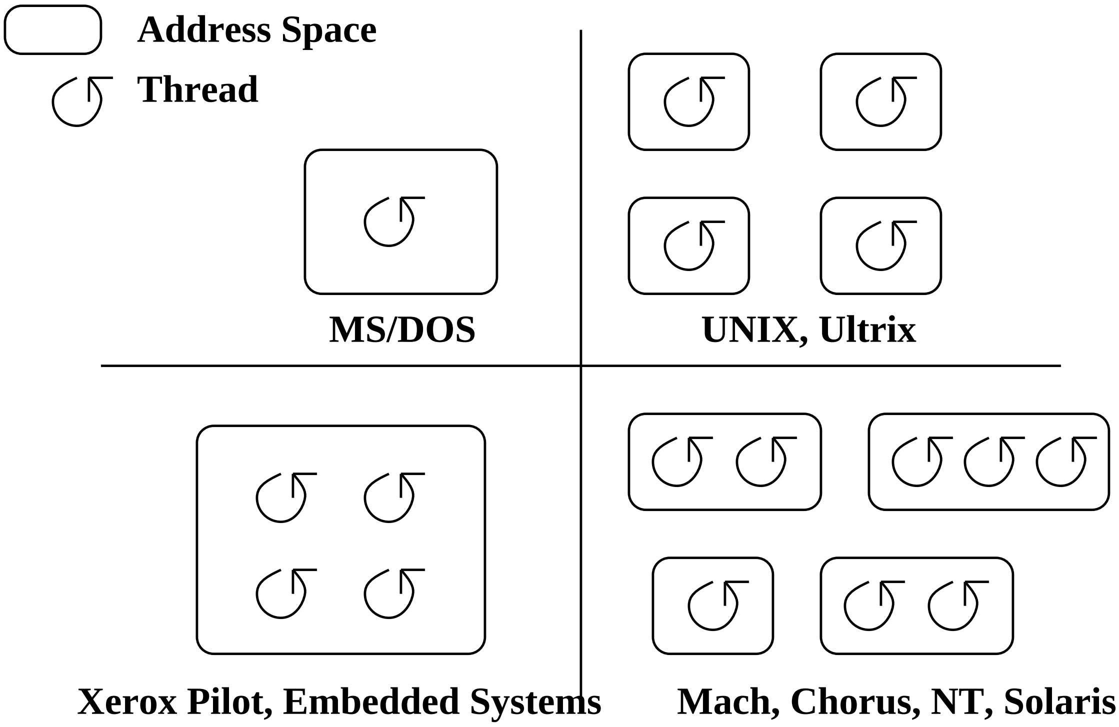
A process defines the address space, text, resources, etc.,

A thread defines a single sequential execution stream within a process (PC, stack, registers). Note, these are the same pieces of state that must be stored and restored during a context switch.

- Threads are bound to a single process.
- Each process may have multiple threads of control within it.
- Since the address space of a process is shared among all the threads it creates, the program does not need to make system calls to cooperate among threads and is thus simpler than message passing and explicitly declared shared-memory counterparts. (Creating a thread may however require a system call.)

Classifying Threaded Systems

Operating Systems can support one or many address spaces, and one or many threads per address space.



In Linux...

```
PTHREAD_CREATE(3)                                 PTHREAD_CREATE(3)
NAME
       pthread_create - create a new thread
SYNOPSIS
       #include <pthread.h>

       int  pthread_create(pthread_t  *  thread, pthread_attr_t *
       attr, void * (*start_routine)(void *), void * arg);

DESCRIPTION
       pthread_create creates a new thread of control  that  exe
       cutes concurrently with the calling thread. The new thread
       applies the function start_routine passing it arg as first
       argument.  The new thread terminates either explicitly, by
       calling pthread_exit(3), or implicitly, by returning  from
       the  start_routine function. The latter case is equivalent
       to calling pthread_exit(3) with  the  result  returned  by
       start_routine as exit code.

       The  attr  argument  specifies  thread  attributes  to  be
       applied to the new thread. See pthread_attr_init(3) for  a
       complete  list of thread attributes. The attr argument can
       also be NULL, in which case default attributes  are  used:
       the  created  thread  is  joinable  (not detached) and has
       default (non real-time) scheduling policy.

RETURN VALUE
       On success, the identifier of the newly created thread  is
       stored in the location pointed by the thread argument, and
       a 0 is returned.  On  error,  a  non-zero  error  code  is
       returned.
  ...............
AUTHOR
       Xavier Leroy <Xavier.Leroy@inria.fr>
SEE ALSO
       pthread_exit(3),    pthread_join(3),    pthread_detach(3),
       pthread_attr_init(3).
                           LinuxThreads                         1
```

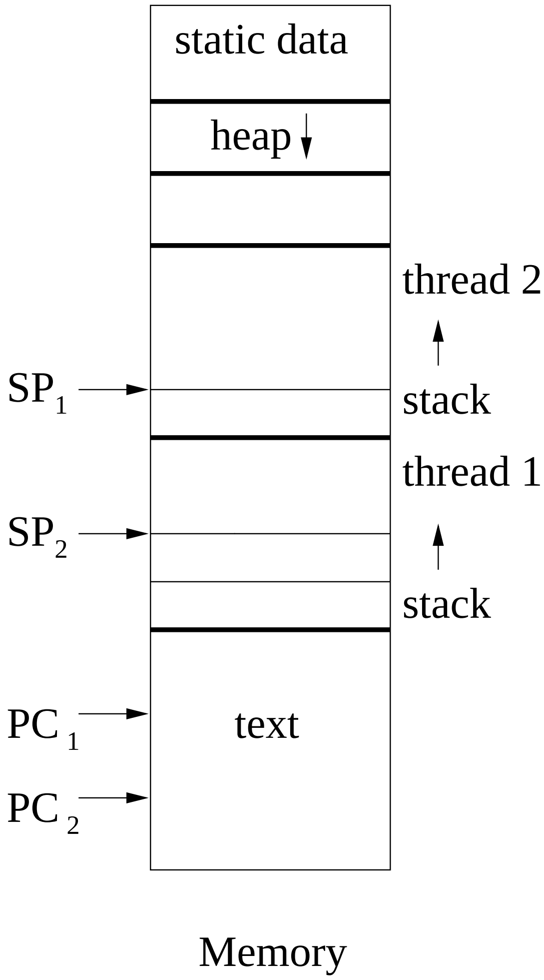
Example Threaded Program

```
main()
    global in, out, n, buffer[n];
    in = 0;
    out = 0;
    fork_thread (producer());
    fork_thread (consumer());
end

producer
    repeat
        . . .
        nextp = produced item
        while in+1 mod n = out do no-op
        buffer[in] = nextp
        in = in+1 mod n

consumer
    repeat
        while in = out do no-op
        nextc = buffer[out]
        out = out+1 mod n
        . . .
        consume item nextc
```

One possible memory layout:



Forking a thread can be a system call to the kernel, or a procedure call to a thread library (user code).

Processes and Thread Summary

- Processes can communicate either with message passing or shared memory.
- Threads differ from forked processes in that they share the same address space, while forked processes simply start with a copy of the same address space.
- Each thread has its own control state (program counter, register values, and execution stack).
- It is much easier to communicate between threads than between processes.
- On the other hand, it is easy for threads to inadvertently disrupt each other since they share the entire address space. (Protection?)