Processes

Creating processes

- Unix: **`fork()`** + **`exec()`**
- Windows: **`CreateProcess()`**

Detecting process termination

- Unix: **`wait()`**
- Windows: **`GetExitCodeProcess()`**

Today: process communication

Starting A Processes

`fork.c`

`fork_parent.c` and **`fork_child.c`**

Process Control

- What happens if a parent exits before a child?
- Can a process terminate another process?
- Who cleans up?

Communication

How can processes communicate?

- Command-line arguments
- Files
- Pipes
- Sockets
- Messages
- Shared memory

File Descriptors

```
fgets(buffer, n, stdin)
```

⇒ `read(0, buffer, n)`

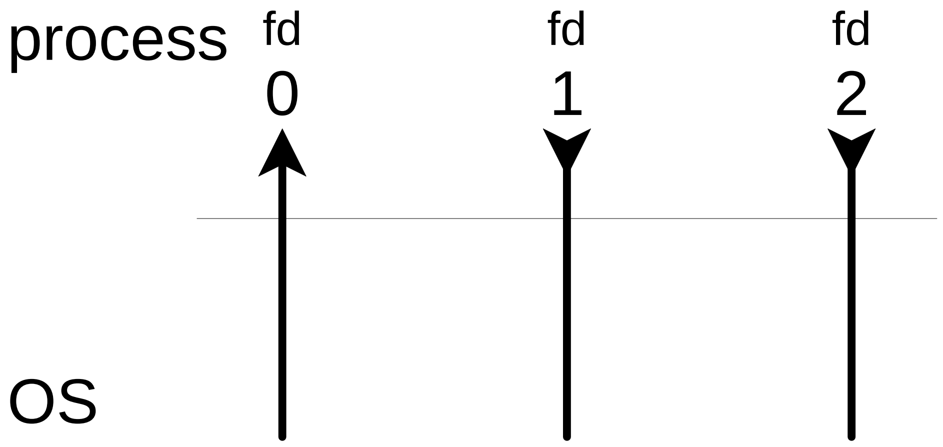
```
printf("hi")
```

⇒ `write(1, "hi", 2)`

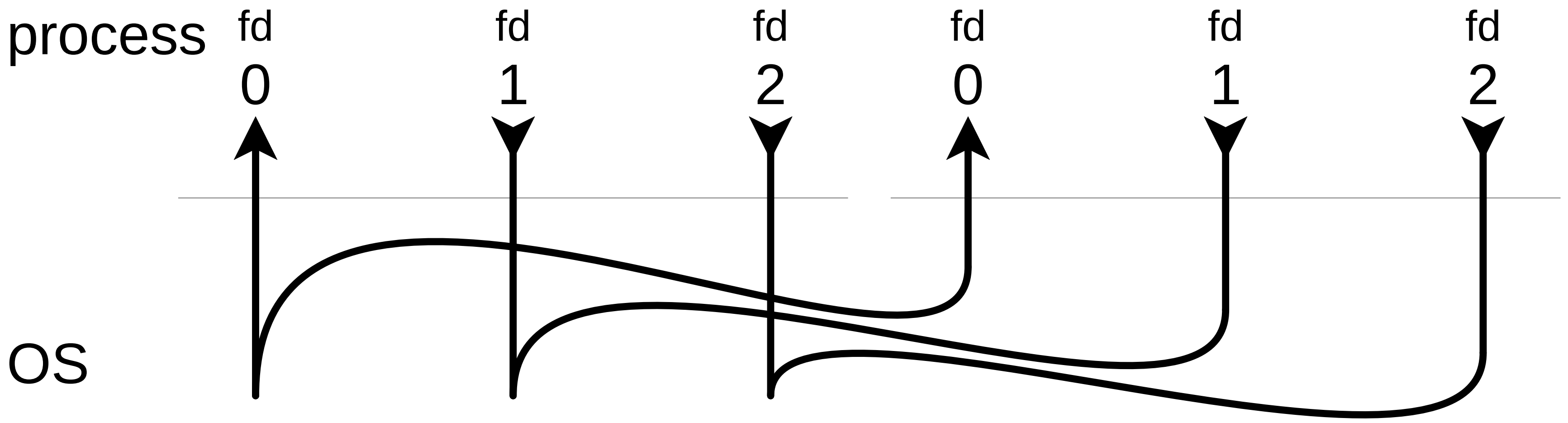
```
fprintf(stderr, "hi")
```

⇒ `write(2, "hi", 2)`

File Descriptors

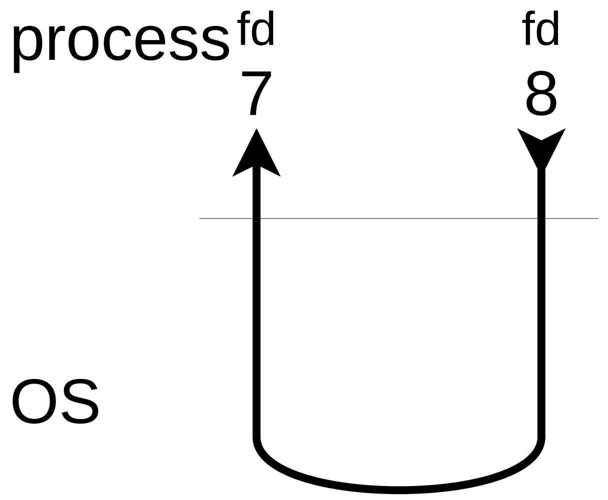


File Descriptors



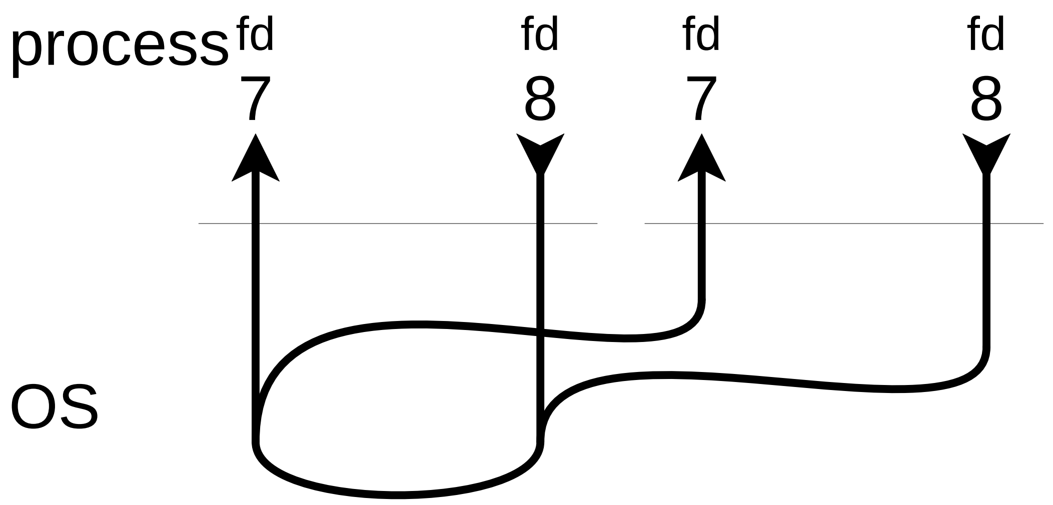
Pipes

```
int fds[2];
pipe(fds);
```



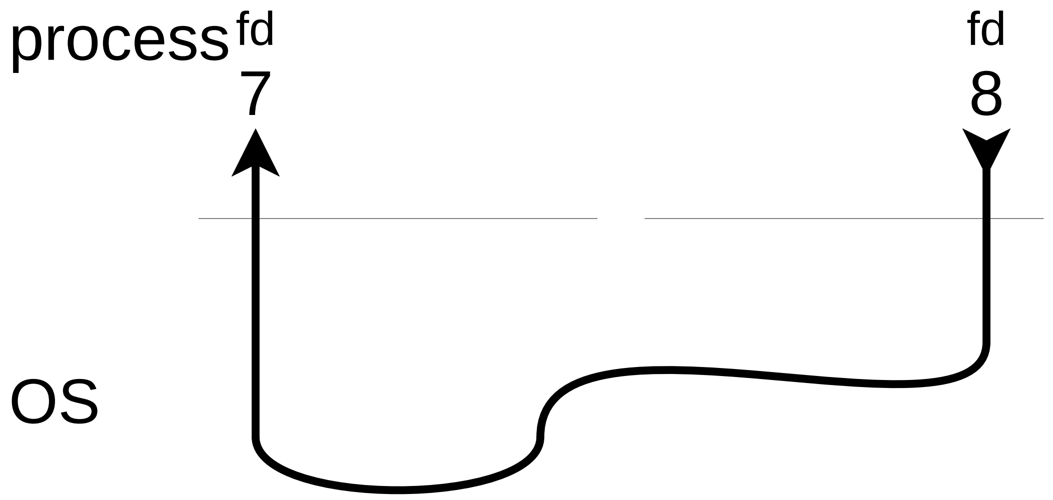
Pipes

```
int fds[2];
pipe(fds);
```



Pipes

```
int fds[2];
pipe(fds);
```



Using Pipes

`pipe()` and **`dup2()`**:

`pipe_parent.c` and **`pipe_child.c`**

Buffering

Beware of pipe buffer limits

`echo_parent.c` and **`echo_child.c`**

Message Passing

```
main() {
  ...
  q_id = msgget();
  if (fork() != 0)
    producer();
  else
    consumer();
}
```

```
producer() {
  while (1) {
    ...
    produce item nextp
    ...
    msgsnd(q_id, nextp);
  }
}
```

```
consumer() {
  while (1) {
    msgrcv(q_id, nextp);
    ...
    consume item nextp
    ...
  }
}
```

can also use **`msgtok(ftok())`** after **`fork()`**

Message Passing vs. Pipes

Can you build message passing on top of pipes?

Shared Memory

```
main() {
  ...
  buffer = shmget();
  if (fork() != 0)
    producer();
  else
    consumer();
}
```

```
producer() {
  while (1) {
    ...
    produce item nextp
    ...
    while (((in+1) % n) == out);
    buffer[in] = nextp;
    in = (in+1) % n;
  }
}
```

```
consumer() {
  while (1) {
    while (in == out);
    nextp = buffer[in];
    out = (out+1) % n;
    ...
    consume item nextp
    ...
  }
}
```

can also use **`shmget(ftok())`** after **`fork()`**

Lots of problems with this code...

Shared Memory vs. Message Passing vs. Pipes

Can you build shared memory on message passing?
On pipes?

How about the other direction?

Etc.

- Direct vs. indirect messages
- Sockets
- RPC

From Processes Threads

For processes to cooperate:

- Create several processes
- Arrange a way to share data
- Context switch back and forth between the processes

Unfortunately, these operations are relatively *inefficient* for the machine and *inconvenient* for programmers.

From Processes Threads

What is shared between processes?

- **Code** is shared if we never call **exec()**
- **Data** is shared since that’s the point
- **Privileges** are shared
- **Resources** are shared (open files and sockets)

What is not shared?

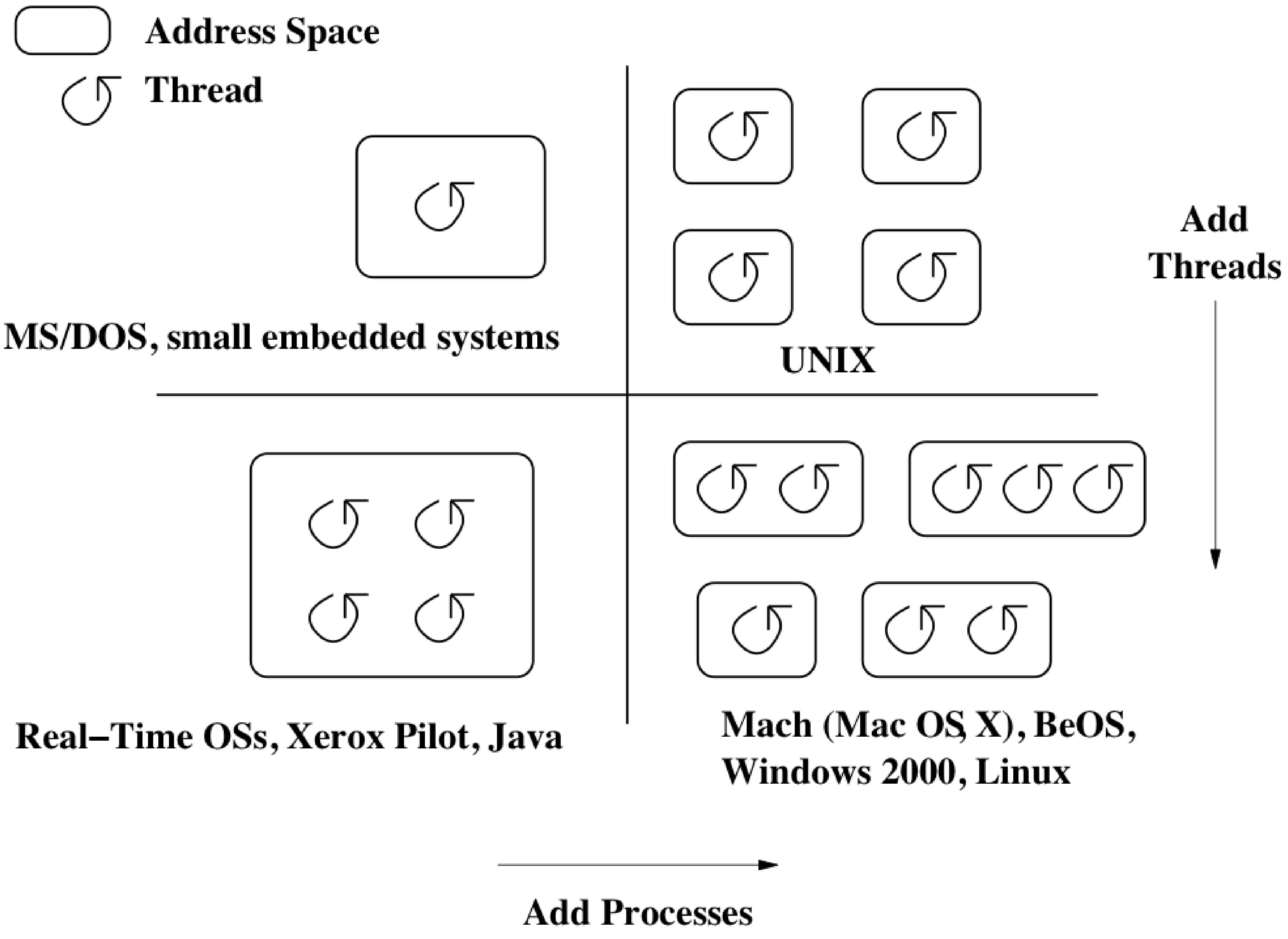
- **Execution state**: PC, SP, registers

Threads

Key idea: Separate the concept of a process from its execution context.

- **Process**: address space, privileges, kernel resources
- **Thread**: execution state (PC, SP, registers)

Processes and Threads


Address Space
Thread
MS/DOS, small embedded systems
UNIX
Add Threads
Real–Time OSs, Xerox Pilot, Java
Mach (Mac OS X), BeOS, Windows 2000, Linux
Add Processes

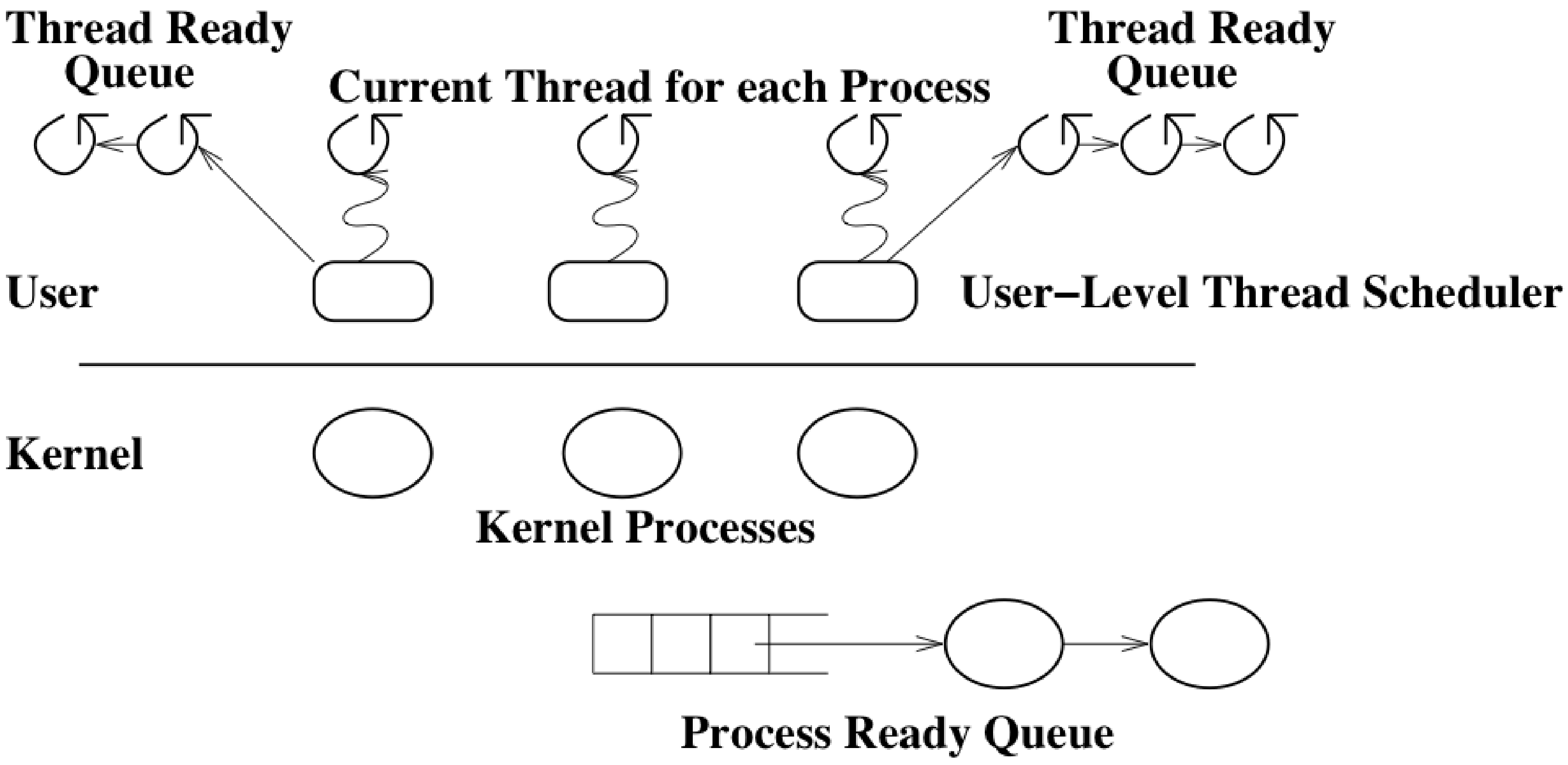
Threads

- Every thread belongs to a particular process
- Processes are like containers that can hold many threads (and all threads die when their process exits)
- Threads are the unit of CPU scheduling
- Switching between threads in the same process is more efficient than switching between processes

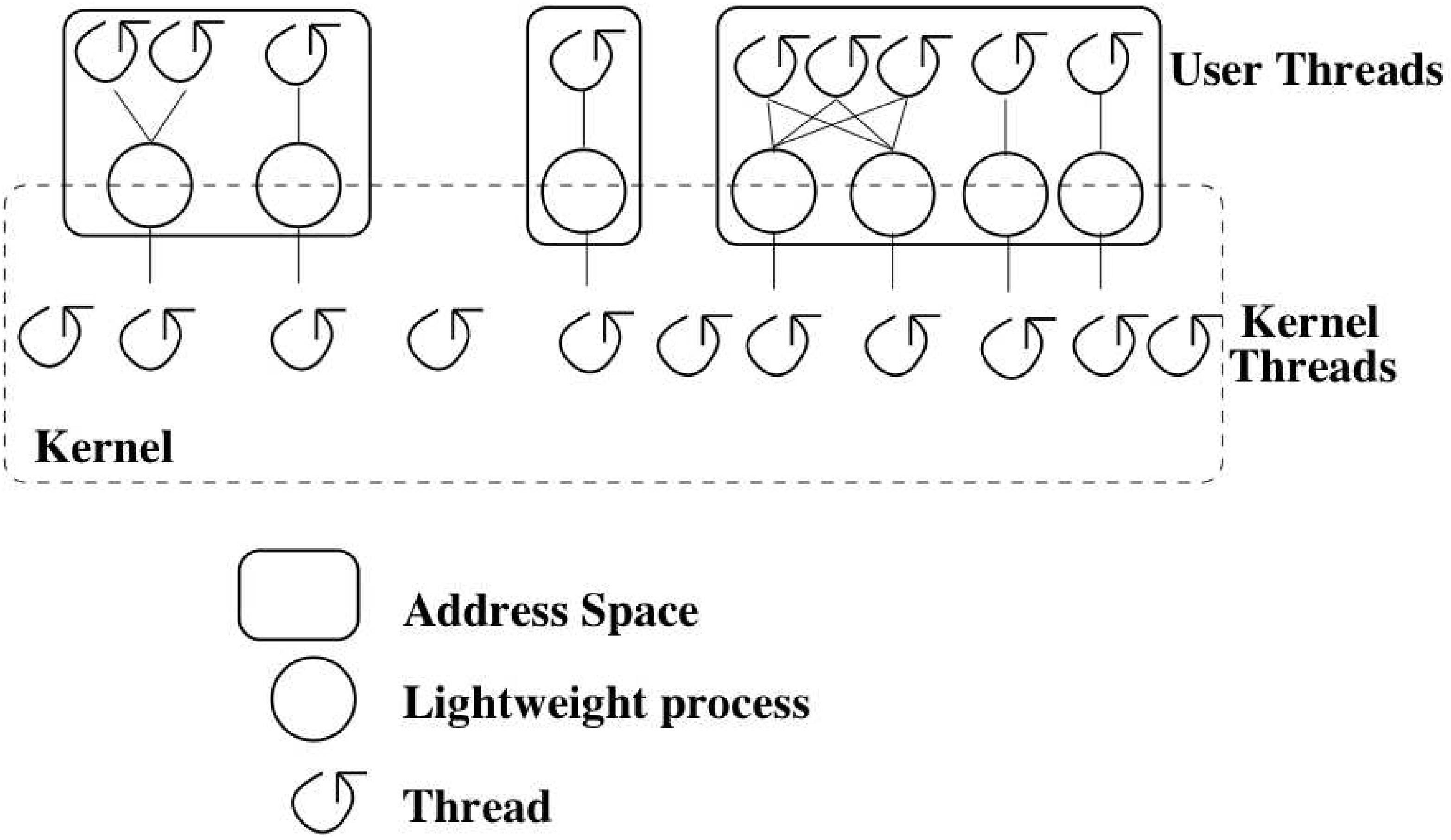
Kernel vs. User Threads

- A **kernel thread** is a thread that the OS knows about
    - a.k.a. **lightweight process**
    - OS schedules threads instead of processes
- A **user thread** is a thread that the OS doesn’t know about
    - Faster and more flexible in principle
    - Not in parallel, problems with blocking system calls
- A mixture $\Rightarrow$ many-to-many models

User Threads



Kernel Threads



Using Threads

```
item buffer[n];

main() {
  ...
  pthread_create(producer);
  consumer();
}
```

```
producer() {
  while (1) {
    ...
    produce item nextp
    ...
    while (((in+1) % n) == out);
    buffer[in] = nextp;
    in = (in+1) % n;
  }
}
```

```
consumer() {
  while (1) {
    while (in == out);
    nextp = buffer[in];
    out = (out+1) % n;
    ...
    consume item nextp
    ...
  }
}
```

Threads for Now

For now, it’s enough to know that threads exist.

Later, we’ll get back to this topic, along with the synchornization tools that you need to make it work.

New Homework

HW2:

Implement a simple Unix shell