Sheets for MIEIC's SOPE

based on teaching material supplied by
A. Tanenbaum for book:
Modern Operating Systems, ed...

Chap 2: Processes

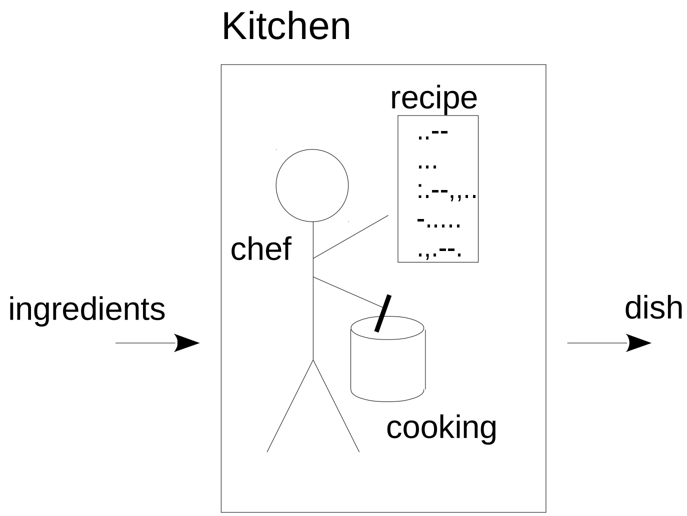
Chapter 2 -1

Processes

Processes
Threads
Interprocess communication (part 1)

Processes

Process *vs* Program

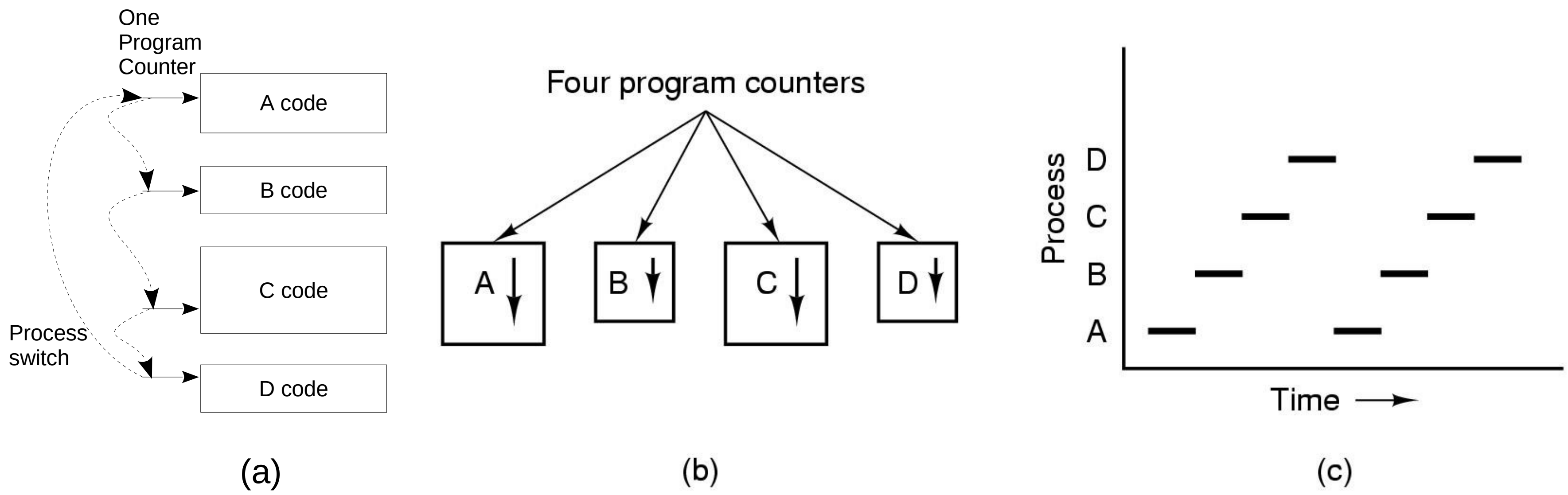


The making of a dish!

Pair the terms in Kitchen to:

- computer
- processor
- process
- program
- input
- output

Processes
The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Process data

Attribute/resource	**Meaning/Info**
PID	Process IDentifier
PPID	Parent Process IDentifier
real U/GID	User/Group IDentifier of who initiated the process
current dir	directory to where names of files are referenced by default
file descriptor table	info on open files; descriptor is table index
environment	initially inherited from parent process
text space	memory where program instructions lie; read only
stack space	memory automatically managed
heap space	memory managed by the user in runtime
priority	info for process scheduling
signal disposition	masks for delivery or blocking of signals
umask	mask that restrains files' permissions on creation

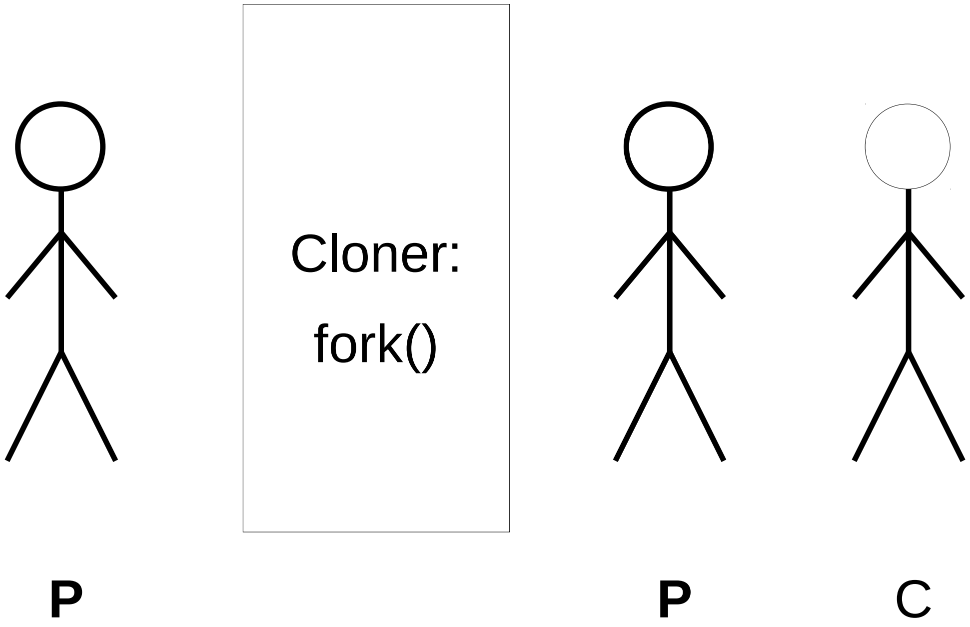
Typical information pertaining to a process

saved and retrieved on preemption and re-scheduling

Process Creation

Principal events that cause process creation

- system initialization
- user/system request (e.g. Unix's `fork()`)



Process Creation (2)

Unix's fork()

- almost a clonage (same code, data, open files...)
- but: different, independent processes (≠ PID, PPID,...)

```
printf("I am the parent!");
int id = fork();
switch (id) {
  case -1:  perror ("fork"); exit (1);
  case 0:   printf("I am the child!"); break;
  default:  printf("I am the parent of: %d", id);
  }
printf("I am the parent or the child!");
```

Process Termination

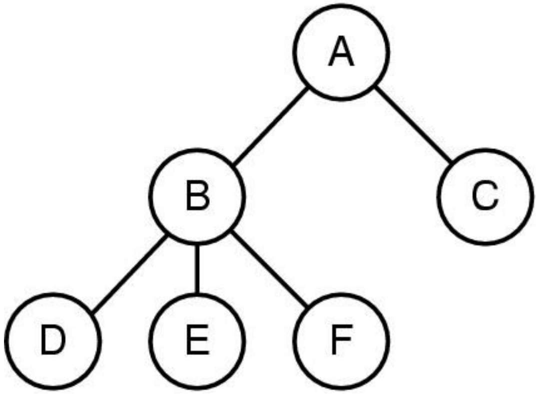
Conditions which terminate processes

- normal exit (voluntary)
- error exit (voluntary)
- fatal error (involuntary)
- killed by another process (involuntary)

Process Hierarchies

Parent creates a child process;
child processes can create their own process;
they form a hierarchy

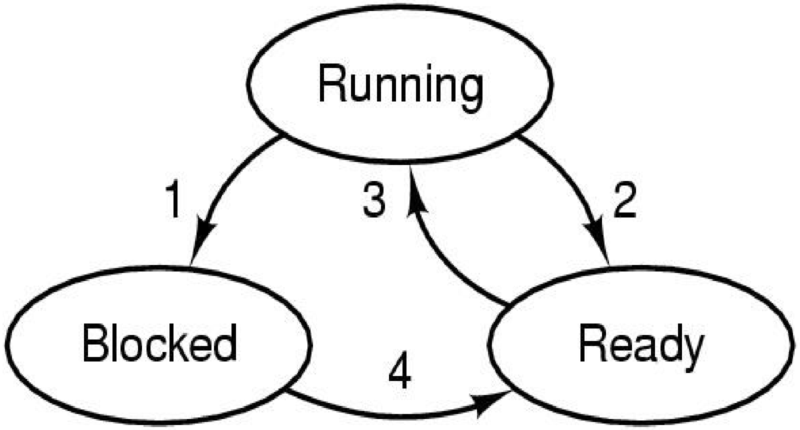
UNIX calls this
a "process group"



MsWindows has no concept of process hierarchy

all processes are created equal

Process States (1)



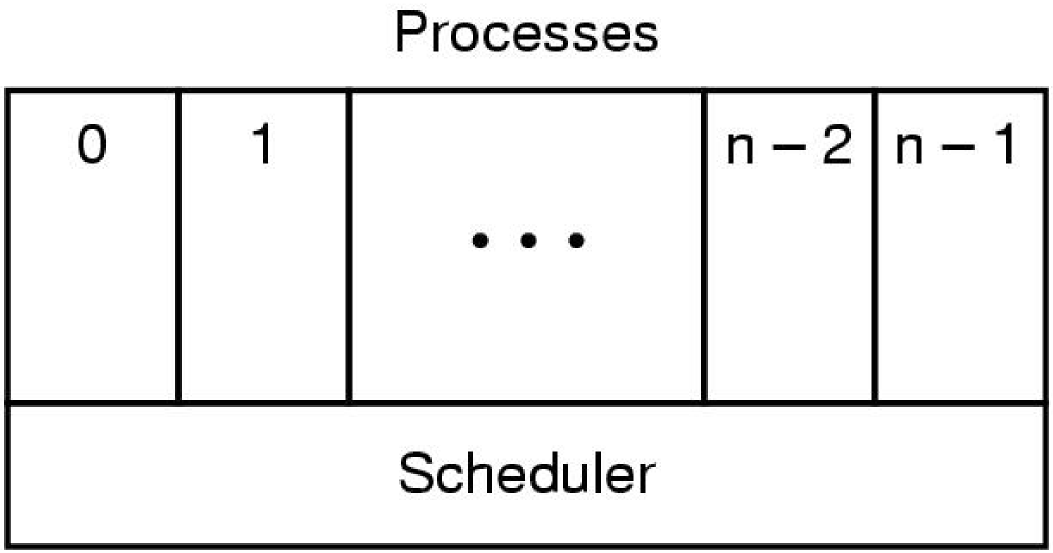
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Possible process states

– running
– blocked
– ready
– (see exercises for more)

Transitions between states: as shown

Process States (2)



Lowest layer of process-structured OS

handles interrupts, scheduling

Above that layer are sequential processes

Implementation of Processes (1)

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children’s CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Typical fields of a Process Table entry

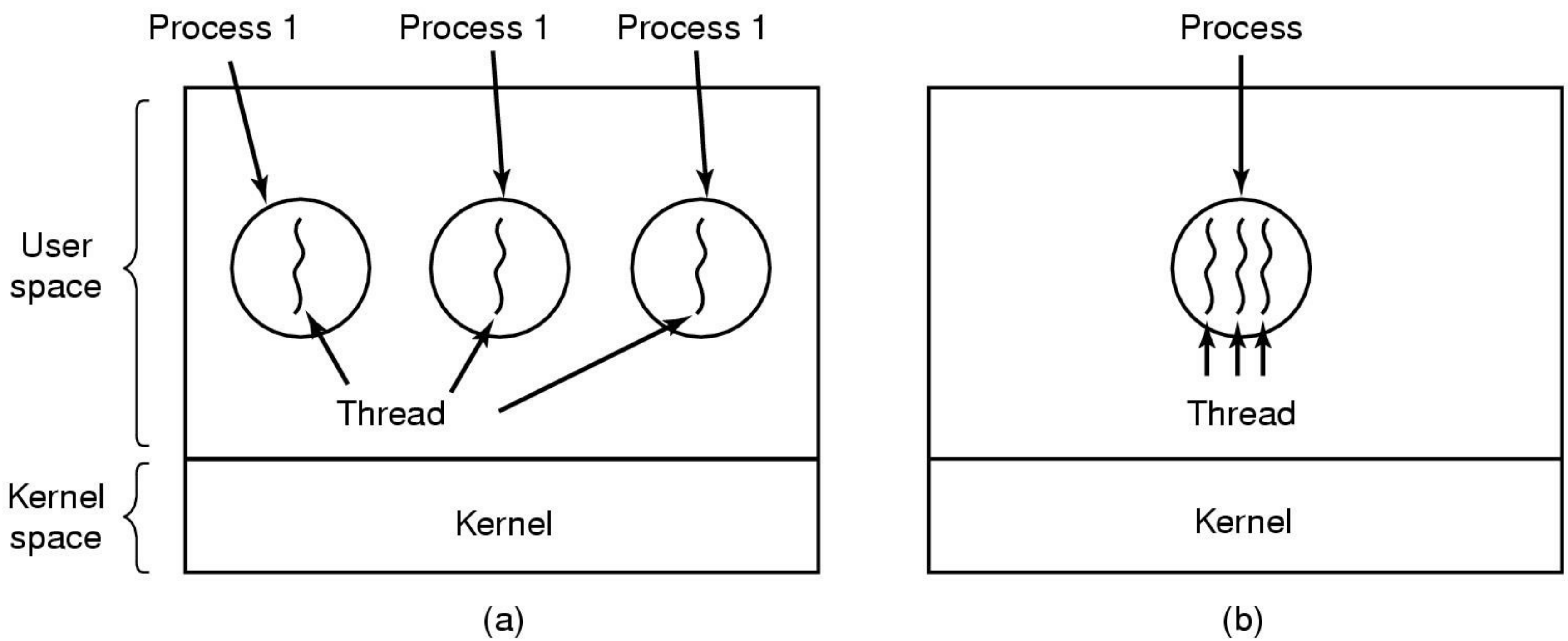
Implementation of Processes (2)

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs

Threads

The Thread Model (1)



(a) Three processes each with one thread

(b) One process with three threads

The Thread Model (2)

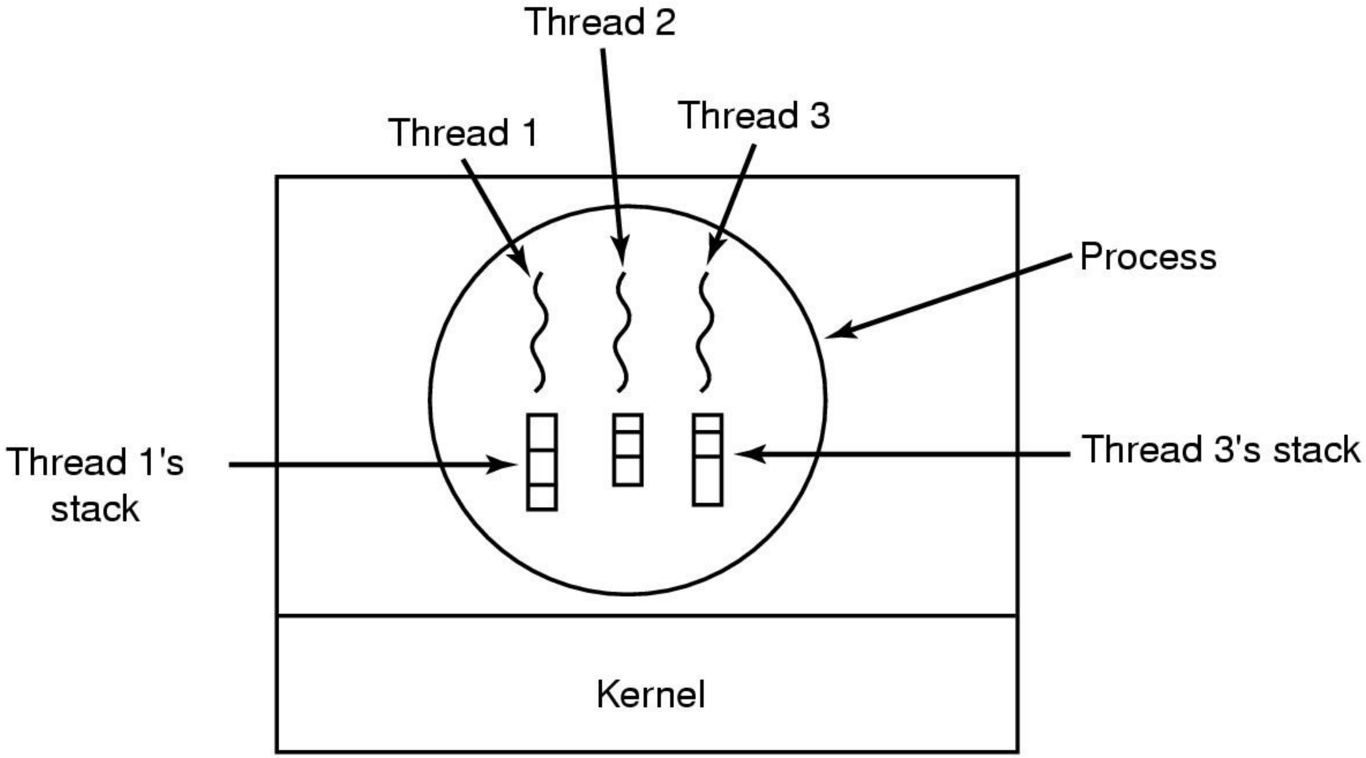
Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Items shared by all threads in a process

vs

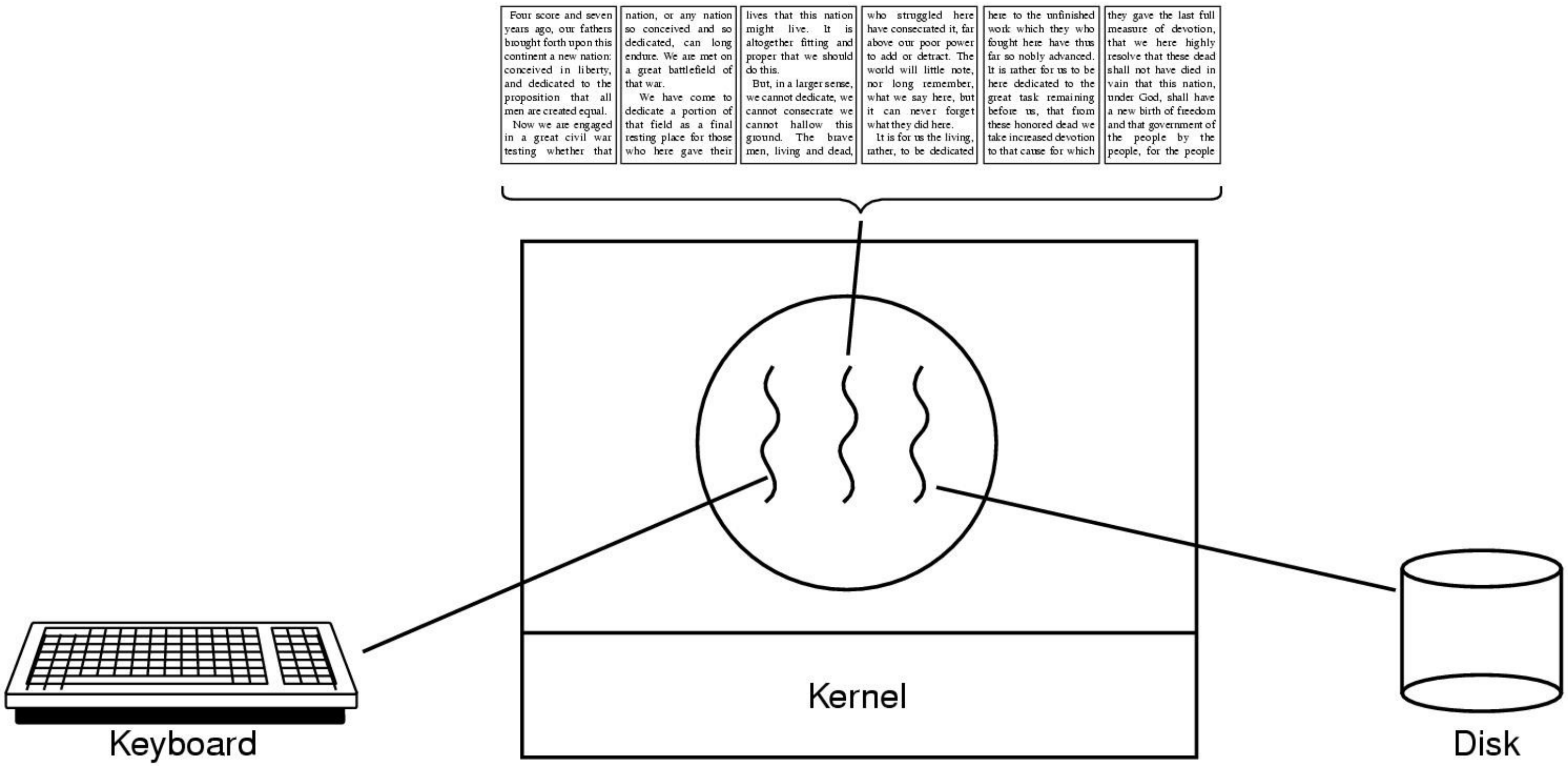
Items private to each thread

The Thread Model (3)



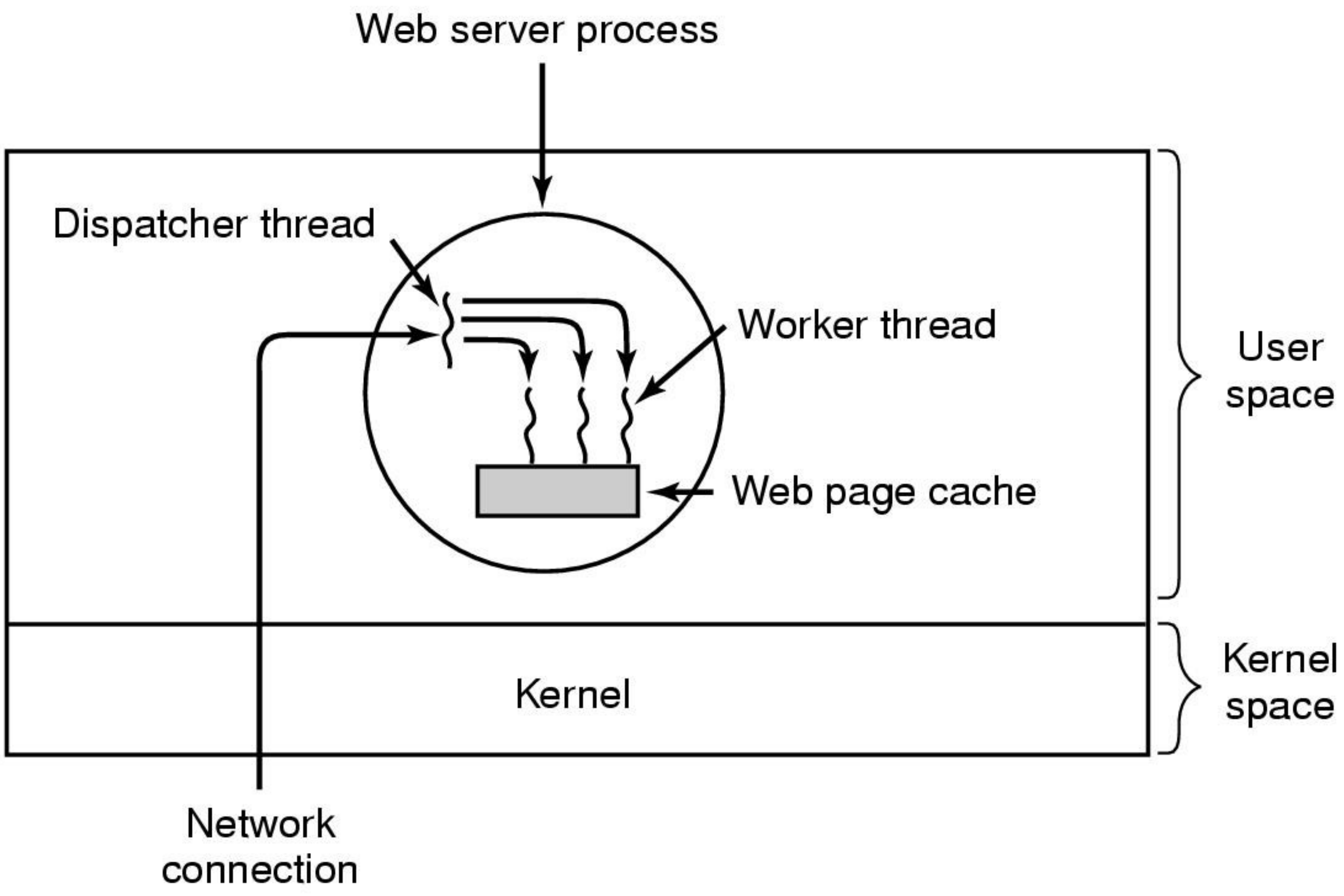
Each thread has its own stack

Thread Usage (1)



Example: a word processor with three threads

Thread Usage (2)



Example: a *multithread* Web server

Thread Usage (3)

Rough outline of code for previous slide

(a) Dispatcher thread

(b) Worker thread

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
       read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

Thread Usage (4)

Three ways to design a high-performing server

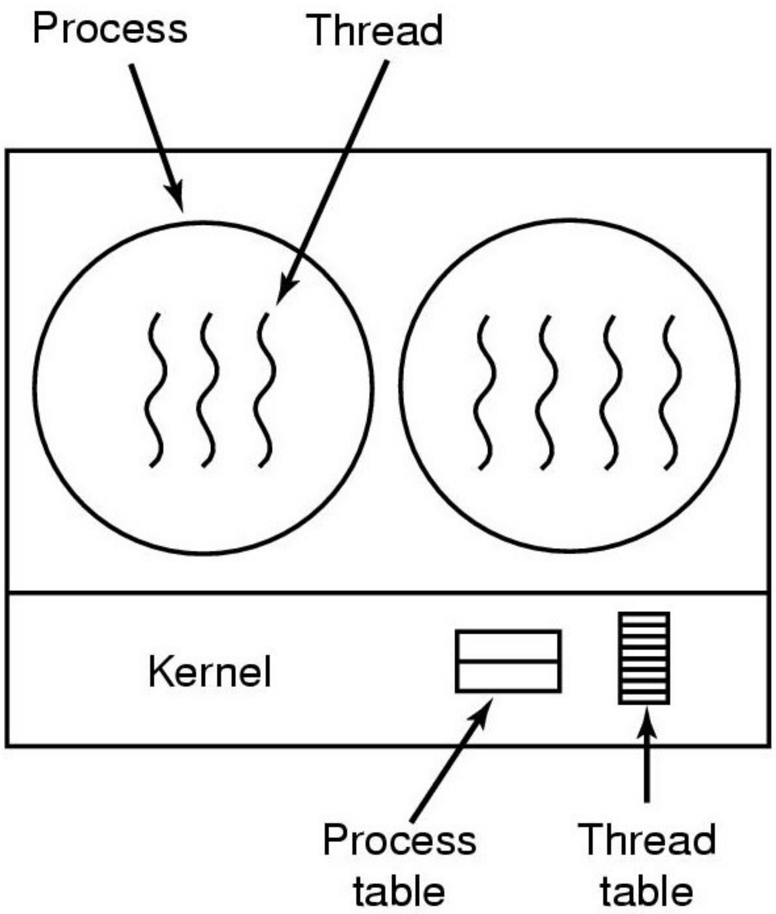
(compared to the most classical way)

Process Model	***Characteristics***			
	Parallelism	***Blocking syscalls***	***Signals or interrupts***	***Data-space***
Most classical	No	**Yes**	**No need**	single
Multi-process	**Yes**	**Yes**	**No need**	independent
Finite-state machine	**Yes**	No	Yes	shared
Multi-thread	**Yes**	**Yes**	**No need**	shared

Posix Thread Creation...

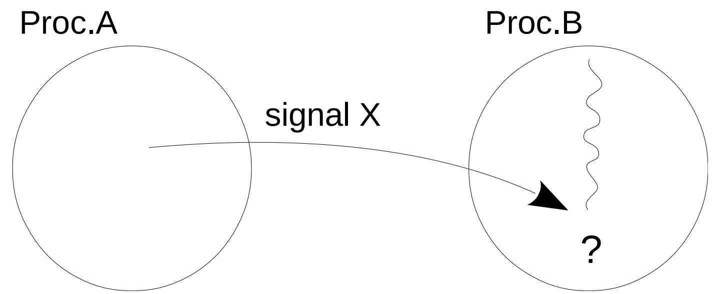
```
main () {
      pthread_t id;
      int err;
   if ((err = pthread_create(&id, NULL, func, NULL)) != 0) {
      fprintf(stderr, "Main thread: %s!\n", strerror(err));
      exit(-1);
   }
   if ((err = pthread_join(id, NULL)) != 0)
      fprintf(stderr, "Main thread: %s!\n", strerror(err));
}

void *func(void *a) {
   printf("New thread id: %lu.\n", (unsigned long) pthread_self());
   pthread_exit(NULL);
}
```

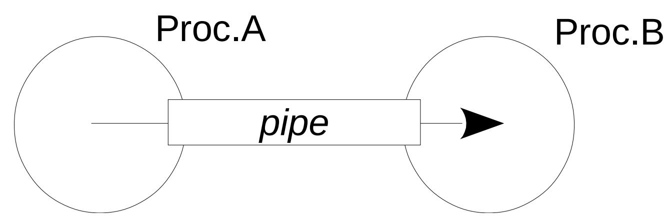
Implementing Threads in the Kernel

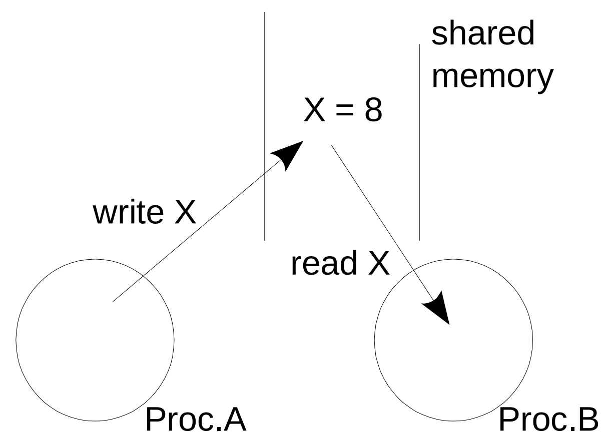


Threads managed by the kernel

InterProcess Communication (IPC)







Processes

- signals
- pipes
- shared memory...

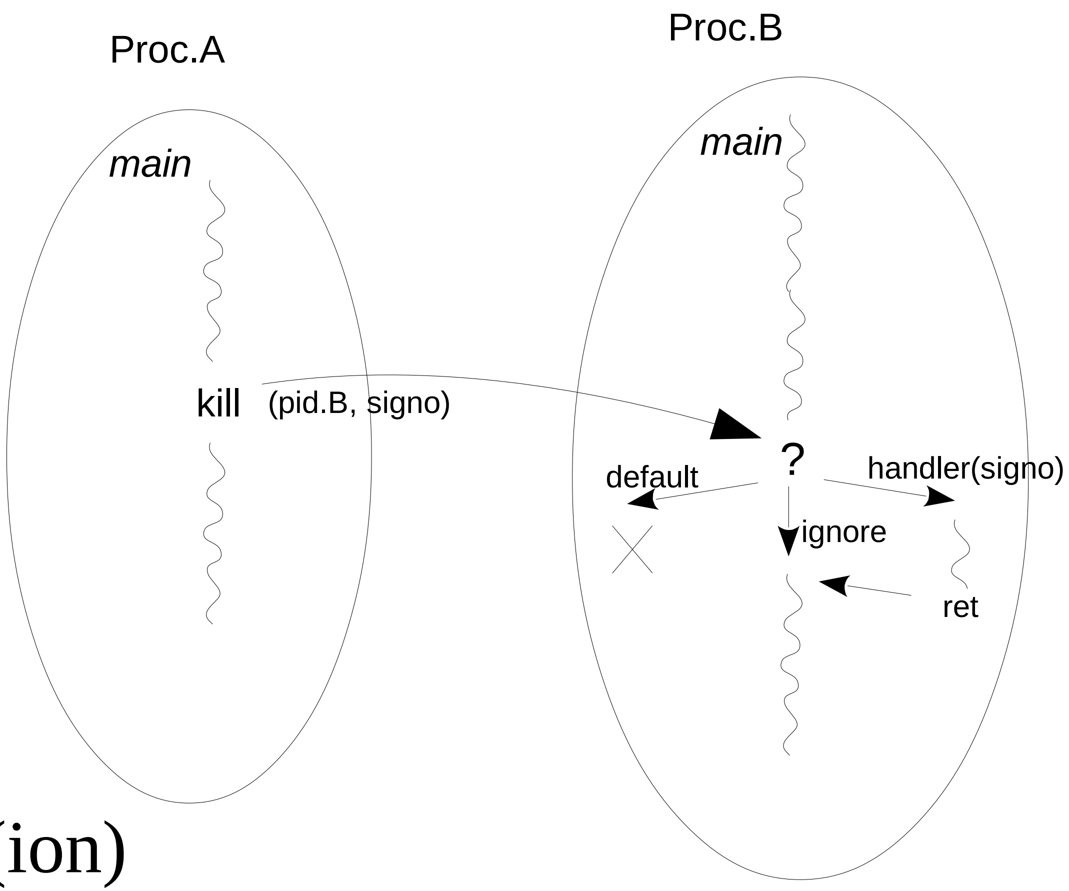
Threads

- shared memory!

Problem:

concurrency -> competition -> races -> deadlocks

...InterProcess Communication...



Signals

signal ≈ interrupt(ion)

kill (pid, signo)

signal handler (signo)

...InterProcess Communication...

Some signals:

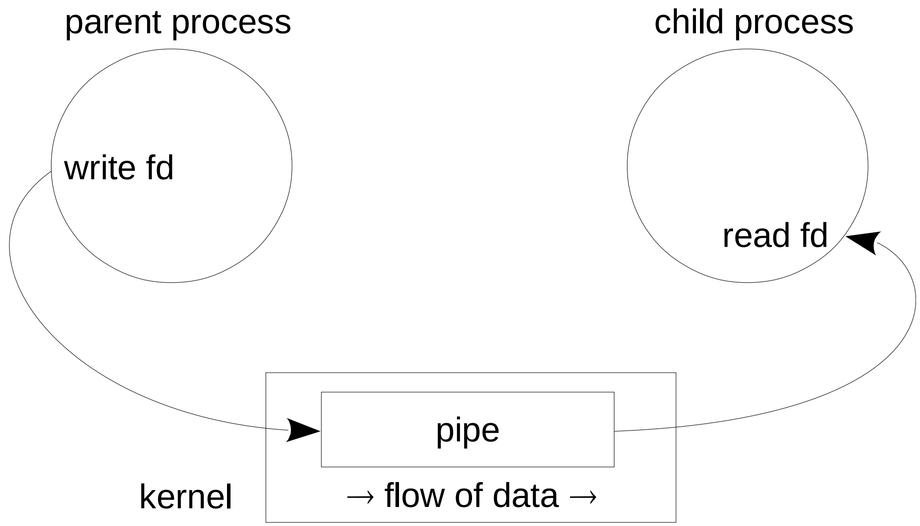
Name	Description	ANSI C	POSIX.1	Default effect
SIGALRM	alarm clock		•	termination
SIGCHLD	state change of child process		*job*	ignored
SIGHUP	terminal hangup		•	termination
SIGINT	terminal interruption	•	•	termination
SIGIO	asynchronous I/O			termination / ignored
SIGKILL	termination no matter what		•	termination
SIGPIPE	no readers in pipe		•	termination
SIGSEGV	invalid memory reference	•	•	termination (*core dump*)
SIGTERM	termination	•	•	termination
SIGUSR1	available for user		•	termination
SIGUSR2	available for user		•	termination

...InterProcess Communication...

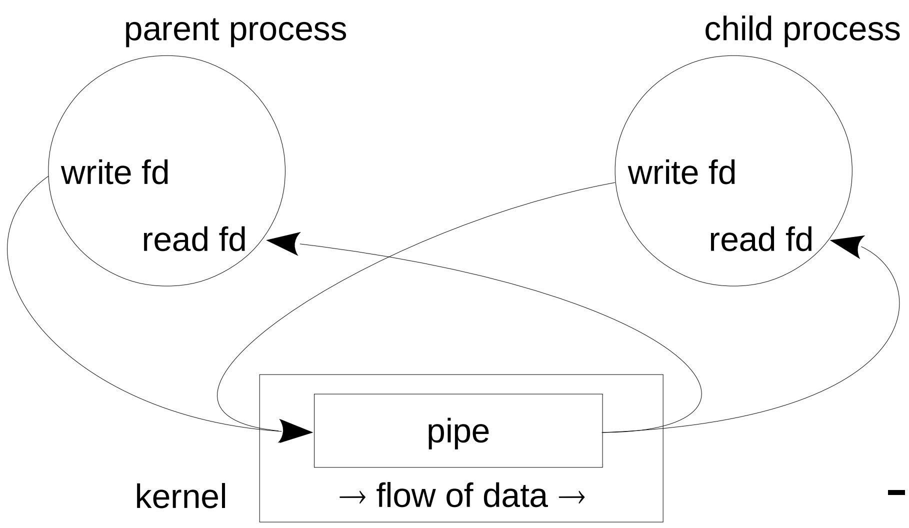
Pipes

parent <–> child

unidireccional



- normal usage



- right after fork()

...InterProcess Communication...

Pipes:

code

```
   int proc;
   int pp[2];
if (pipe(pp) == -1) { perror("pipe()"); exit(1); }
if ((proc = fork()) == -1) { perror("fork()"); exit(2); }
if (proc == 0) {
   close(pp[0]);
   write (pp[1], "Hi, parent!", 1+strlen("Hi, parent!");
   close(pp[1]);
   }
else {                 char msg[1024];
   close(pp[1]);
   read(pp[0], msg, 1024);    // waits...
   printf("Child said: «%s»\n", msg);
   close(pp[0]);
   }
```

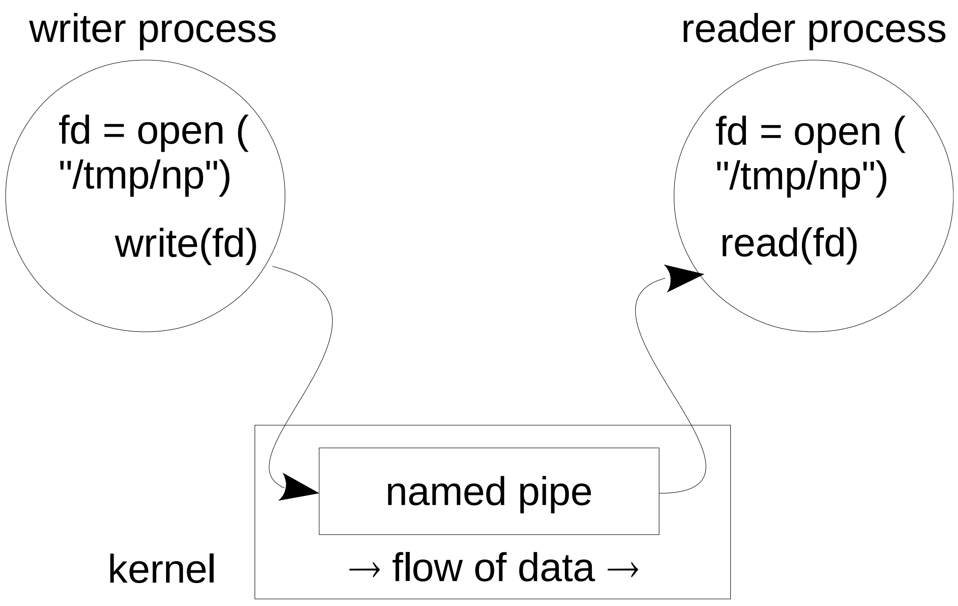
...InterProcess Communication...

Named Pipes (FIFOs)

a process <–> another process

unidireccional

"file" in File System



...InterProcess Communication

Named Pipes:

code

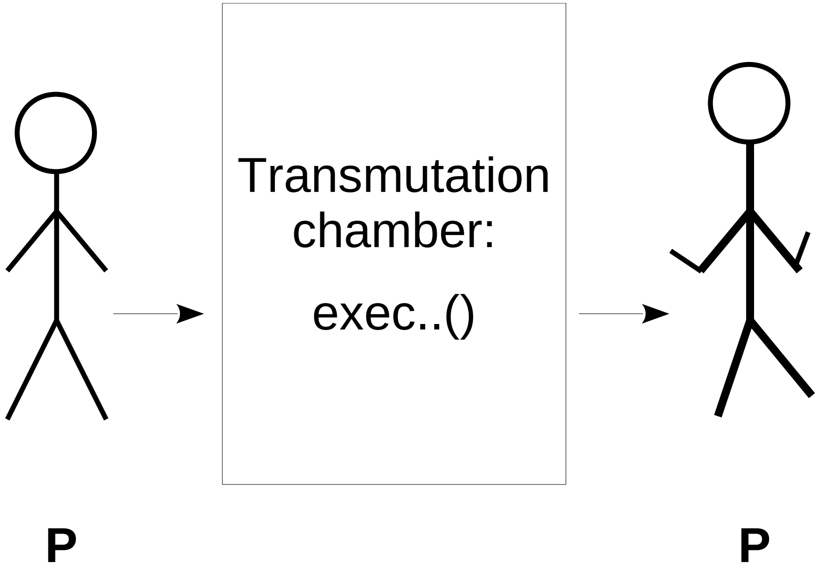
```
READER:
      int np;
      char msg[1024];
   if (mkfifo("/tmp/np",0666) < 0) perror ("mkfifo");
   while ((np = open ("/tmp/np", O_RDONLY)) < 0)
      ;                // synchronization...
   read(np, msg, 1024);        // waits...
   printf("Writer colleague said: «%s»\n", msg);
   close(np);
```

```
WRITER:
      int np;
   if (mkfifo("/tmp/np",0666) < 0) perror ("mkfifo");
   while ((np = open ("/tmp/np", O_WRONLY)) < 0)
      ;                // synchronization...
   write (pp[1], "Hi, reader colleague!", 1+strlen("Hi, parent!"));
   close(np);
```

Annex: Starting a new program: exec()

Process changes its running code

- keeps identification (PID) & some few things
- Unix's `execve()` or related library function e.g. `execlp()`)



...Starting a new program: exec()

Exec():

code

```
    char *cmdline[] = { "sleep", "30", (void *)0 };
    char *newenviron[] = { "USER=Linus", NULL };

if(fork() == 0) { // child
    printf("\nChild: %d. USER=%s", getpid(), getenv("USER"));
    if (execve("/bin/sleep", cmdline, newenviron) == -1)
        perror("execve");
    exit (1); // if execve fails...
    }
else // parent
    ...
```

Exercise: change this code to show that USER was changed after the exec...