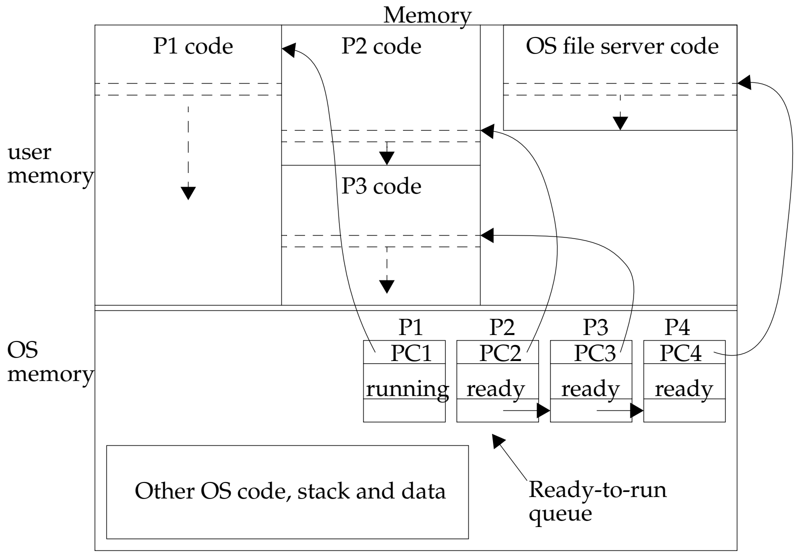
Process: a program in execution.

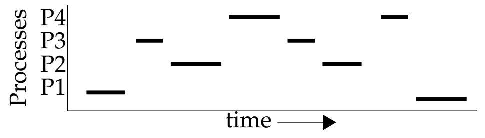
Programs are code. Processes are code plus input, output and a state.

2.1.1 Process Model

Most OS's can run several sequential processes **concurrently** (interleaved in time).

- At any instant, the CPU is running only one process.
- The rapid switching back and forth between processes is called **multiprogramming**.
- Each process has its own virtual CPU. In other words, each process is unaware it is sharing the CPU with other processes.





Processes will run for different amounts of time each making progress in **pseudoparallel**.

If the same four programs are run again, it is likely that they will be run in different order and for different amounts of time.

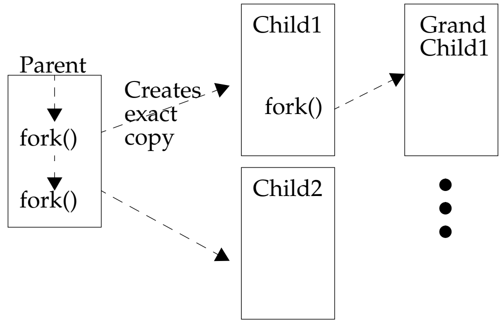
Therefore, programs must NOT be programmed with built-in assumptions about timing ! Why ?

Process Hierarchies

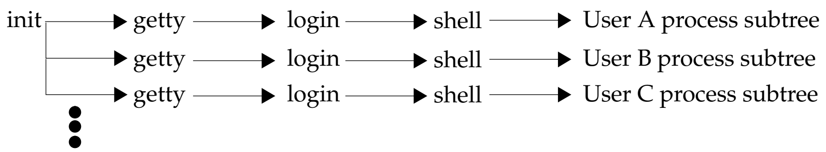
Most *real* OS's provide ability for one process to create others.

In UNIX, the system call is fork().

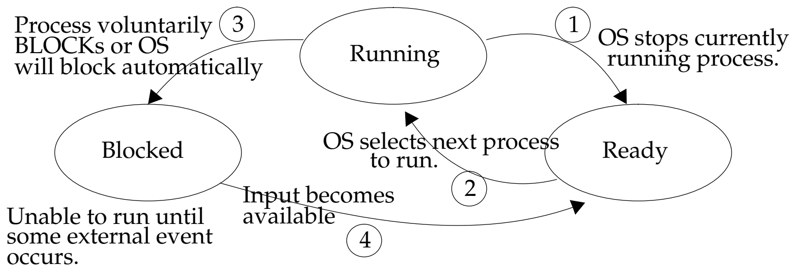
Parent/child relationship.



In MINIX, **init** is parent of all other processes.



Process States



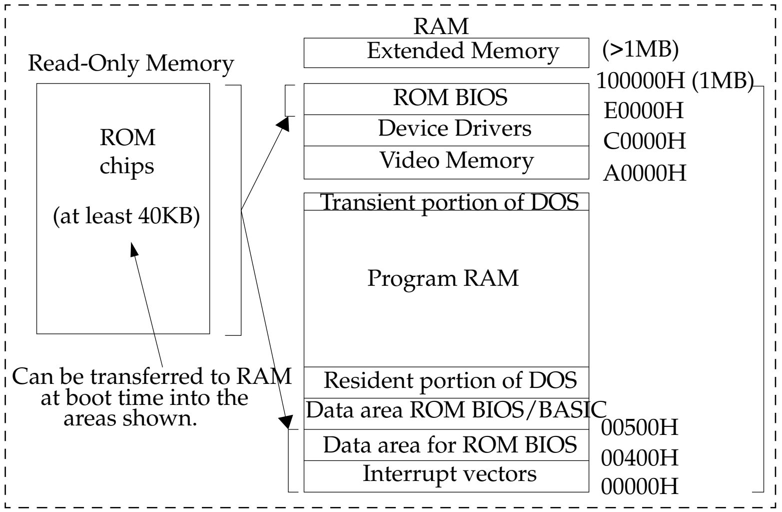
Two different mechanisms for

Transition arc 1: OS may stop a process because it has used its time quantum.

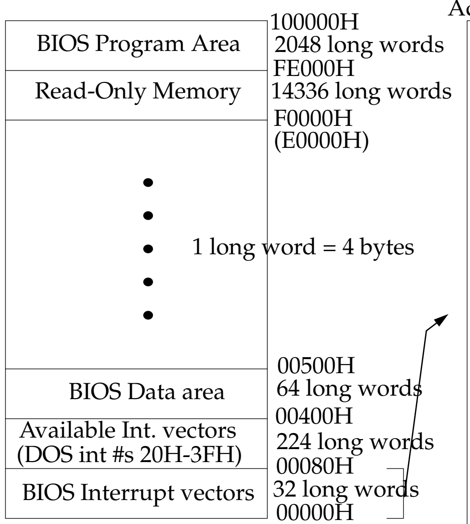
Transition arc 3: Processes block if input is not available.

Hardware Interrupts provide mechanism for transition arcs 1, 2 and 4.
What are interrupts and how do they work ? (OS magic).

DOS Memory Map:

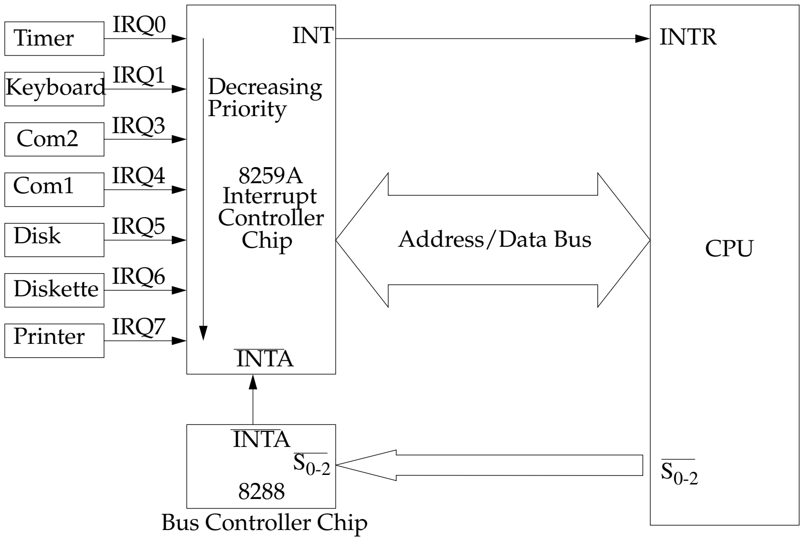


DOS Hardware/Software Interrupts:



Address (Hex)			Interrupt #	
7C-7F	Video Graphic Chars		1FH	Pts to Data
78-7B	Diskette Parameters		1EH	
74-77	Video Initialization		1DH	
70-73	Timer Tick	(18.2/sec)	1CH	
6C-6F	Keyboard Break		1BH	
68-6B	Time of Day		1AH	Software Interrupts Synchronous
64-67	Bootstrap		19H	
60-63	Resident BASIC		18H	
5C-5F	Printer		17H	
58-5B	Keyboard		16H	
54-57	Cassette		15H	
50-53	Communications		14H	
4C-4F	Diskette/Disk		13H	
48-4B	Memory		12H	
44-47	Equipment Check		11H	
40-43	Video		10H	
3C-3F	Printer		FH	Hardware Interrupts Asynchronous 8259A
38-3B	Diskette		EH	
34-37	Disk		DH	
30-33	Communications		CH	
2C-2F	Communications		BH	
28-2B	Reserved		AH	
24-27	Keyboard		9H	
20-23	Time of Day		8H	
1D-1F	Reserved		7H	Microprocessor Interrupts
18-1B	Reserved		6H	
14-17	Print Screen		5H	
10-13	Overflow	(CPU)	4H	
C-F	Breakpoint	(CPU)	3H	
8-B	Non-maskable	(8087)	2H	
4-7	Single Step	(CPU)	1H	
0-3	Divide by zero	(CPU)	0H	

80x86 Hardware Support for Interrupts:



80x86 Hardware Interrupt Processing Sequence:

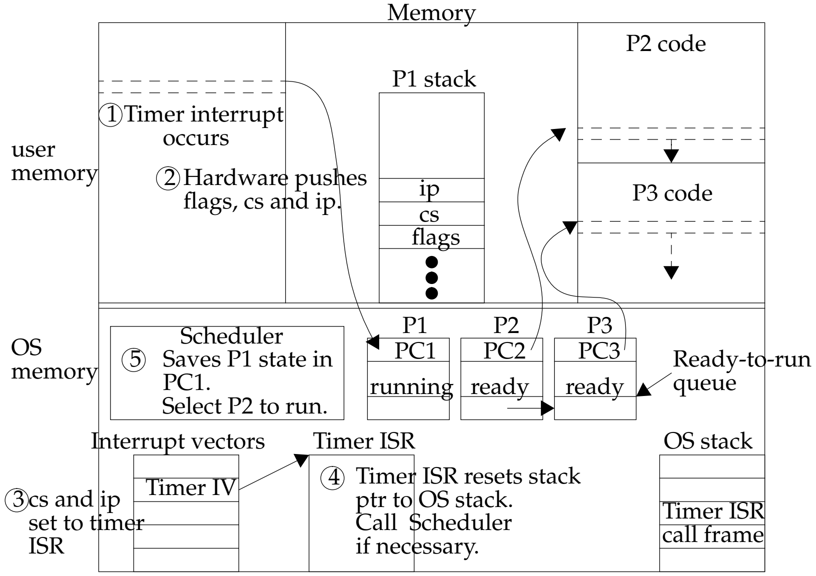
1) Keyboard interrupt occurs (IRQ1) when the user presses a key.
2) If IRQ1 is not masked off and no interrupt is pending on IRQ0, then
3) 8259A raises INTR line of CPU.
4) CPU honors interrupt if **if** flag in **flags** register is set.
5) CPU signals bus controller (8288) via status lines ($\overline{S}_{0\text{-}2}$) to put first int ack on $\overline{\text{INTA}}$ line.
6) CPU reserves BUS for future communications with 8259A.
7) CPU signals bus controller (8288) via status lines ($\overline{S}_{0\text{-}2}$) to put second int ack on $\overline{\text{INTA}}$ line.
8) 8259A puts interrupt number 0x9 onto the Bus.
9) CPU reads the byte 0x9 off of the Bus and multiplies by 4 to get the address of the interrupt vector - in this case it is 0x24.
10) CPU saves the state of the machine by pushing **flags** register onto stack.
11) CPU clears interrupt enable flag (**if**) and the trap flag (**tf**) preventing other interrupts. This will allow the ISR to run with interrupts off until it gets to a point in its execution where it can safely re-enable interrupts.
12) CPU pushes **cs** and **ip** register values onto the stack and loads them with **cs** and **ip** values at the interrupt vector address.
13) **iret** instruction executed as last statement in ISR pops the original **ip**, **cs** and **flags** stack values back into the registers.

Now we have the hardware mechanism necessary to allow multiprogramming.

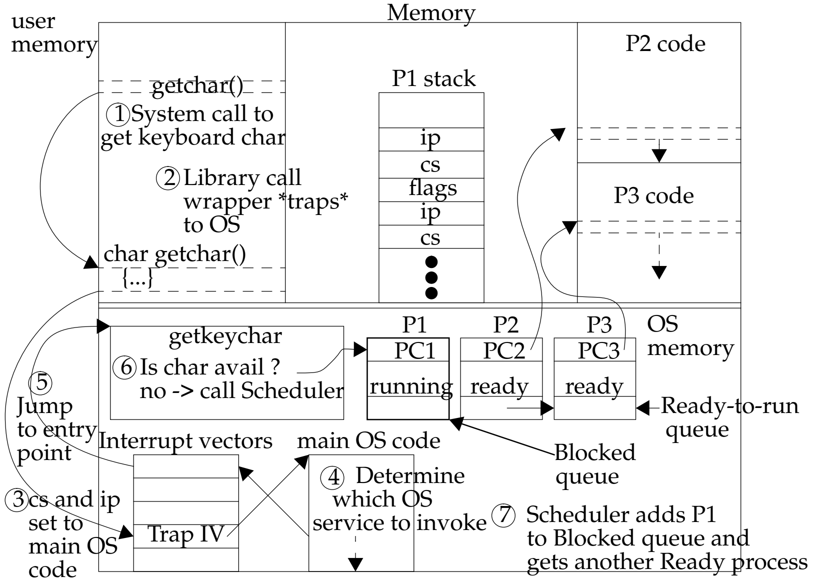
How can we prevent a user process from monopolizing the CPU ?

- Place the interrupt vectors and ISRs in protected memory (accessible only by the OS). Why ?
- Install an Timer ISR that keeps track of how long a process has been running.
- Call the OS Schedular from this routine when a process has used its time slice.

Multiprogramming via the Timer ISR.



Multiprogramming induced by a blocked system call.



P1 will not run again until a keyboard interrupt occurs.

If keyboard interrupt occurs then:
- Keyboard ISR will determine if any process is blocked waiting on input.
- If so, it calls Scheduler to move the blocked process to the Ready queue (Transition arc 4).
- When P1 runs again, it returns from getkeychar OS routine with data and continues.

Therefore:
- Ready and Running states from state transition diagram (slide 5) are similar since the processes in these states are willing to run. Ready state processes -> no CPU is available.
- Blocked state is unique since these processes can NOT run even if the CPU is available.

2.1.2 Implementation of Processes

Process Table: OS maintained table used to save process state.
- One entry per process.
- In MINIX, process management, memory management and file management are handled by separate modules. Process table is partitioned as:

Process	Memory	Files
======	======	====
PC (ip)	ptr to exec	root directory
CPU registers	ptr to data	working directory
stack pointer	ptr to stack	open files
CPU status word (flags)	user id	protection info
process state	process ID	etc.
message queue	process group	
etc.	etc.	

Interrupts and the process model

Rather than thinking about disk interrupts and disk ISRs, we can have devices managed by processes.

In this scheme, a disk controller is simply a process that waits for either:
- A request **message** (e.g. get me block 247)
- A hardware **message** (e.g. block 247 is ready)

Still requires an ISR to send messages on a hardware interrupt.

Advantage: Organizing the OS is much easier.
Disadvantage: There is some overhead associated with messages.

2.1.3 Threads

Traditional Processes:

Single thread of control.
Single PC.

Threads:

Multiple threads of control in one traditional process.
Multiple PCs.

Threads are **lightweight** processes that share some process state info, e.g. same address space and open files.

The OS itself may be organized as a set of threads.

For example, a File server process in which the threads share the same cache.

Advantage: There is less overhead in switching between threads than between full processes.

Threads can be scheduled and managed:

- In user space within the process.
- In kernel space at the OS level.

What are the trade-offs ?

Read the section on the problems associated with threads with respect to OS functions and implementation.

2.2 Interprocess Communication (IPC)

Under certain circumstances, processes need to communicate with each other.

Communication is necessary in client server systems, between processes in a pipeline, in many OS utilities such as print spoolers, etc.

What are the issues ?

- How can one process pass information to another ?
- How can we prevent one process from interfering with another when doing some critical activity ?
- How can we synchronize two or more processes ?

We can pass information between processes using shared memory and a set of synchronization operations.

However, when two or more processes read and write shared memory without a hardware or software protocol to control their access, unpredictable behavior and/or incorrect data results.

2.2.1 Race Conditions

A race condition occurs when

- Two or more processes are reading and writing shared memory AND
- The final result is dependent on the interleaving sequence of the processes.

For example:

Two processes want to print a file.

Print spooler keeps table of filenames to print and a shared variable 'next_slot_avail'.

Process A reads value of 'next_slot_avail' and saves it locally (assume the value is 7).

Timer interrupt occurs and Scheduler runs process B.

Process B reads value of 'next_slot_avail' (7) and saves filename in table at that slot.

Process B updates 'next_slot_avail' to next value (8).

Process A runs again and writes filename into slot 7 (ouch) and overwrites process B's filename at that position in the table.

Process B's file is never printed.

2.2.2 Mutual Exclusion: Race Condition Solution

A mechanism to prohibit more than one process from reading and writing shared memory at the same time OR

A mechanism to provide **mutual exclusion** in **Critical Sections** of the program.

A **Critical Section** (**CS**) is that section of the program code that references/updates shared variables.

Actually, there are four desirable properties that avoid race conditions AND allow concurrent processes to cooperate correctly and efficiently.

1) **Mutual Exclusion**: No two processes may be simultaneously inside their critical sections.

2) **Timing**: No assumptions may be made about the speed or number of CPUs.

3) **Progress**: No process running outside the critical sections may block other processes that want into a critical section.

4) **Bounded Waiting**: No process should have to wait forever to enter its critical section.

2.2.3 Mutual Exclusion via Busy Waiting

Busy waiting: Continuously testing a variable until some value appears.

Solution 1:

Disable interrupts.

Problem 1: User process may never turn interrupts back on again (e.g. process enters an infinite loop) which would bring down the system.

Also, we must give processes access to privledged instructions, allowing them to hog the CPU.

Violates property 3.

Why ?

No timer interrupts -> no multiprogramming possible.

Problem 2: Doesn’t work in a multiprocessor system. Violates property 2.

However, there exist situations in which disabling interrupts is necessary in kernel code, e.g. in ISRs.

Solution 2:

Shared lock variables.

Suppose we have a shared **lock** variable, initially 0 and set to 1 whenever a process is in its critical section. Will the following code fragment provide mutual exclusion ?

```
while ( lock != 0 );
lock = 1;
critical_section();
lock = 0;
```

NO - NOT a solution !!! Violates property 1, e.g.

P0:	P1:	time
determines lock is 0 (context switch)		
	determines lock is 0 sets lock to 1 enters CS (context switch)	
sets lock to 1 enters CS		

Solution 3:

Strict alternation: we allow processes to take turns.

P0:	P1:

`while(TRUE)`
`{`
`while (turn != 0);/*Wait*/`
`critical_section();`
`turn = 1;`
`noncritical_section();`
`}`
	`while(TRUE)`
	`{`
	`while (turn != 1);/*Wait*/`
	`critical_section();`
	`turn = 0;`
	`noncritical_section();`
	`}`

This solution provides mutual exclusion (satisfies property 1).

Problem: Requires processes to know identity of other processes.

Problem: Violates property 3 (progress) since it is possible that turn is 0 (and P0 is executing in noncritical_region()) and P1 wants to enter the critical section. Forces both processes to proceed at speed of slower process.

Also P1 will be permanently locked out if P0 crashes in the noncritical_region() with turn set to 0.