Today: Process Management

- Scheduling (final thoughts)
- Minix Interrupts Handling and IPC
- Process refresher
  - Sequential and concurrent processes
  - Process creation
- Minix Process Creation

Course Roadmap

- Learn about OS through the lens of Minix and Linux
- Linux "mini-course" - 4 lectures
  - L1: Linux architecture, L2: Scheduling, L3: Low-level kernel concepts, L4: Memory management and File Systems

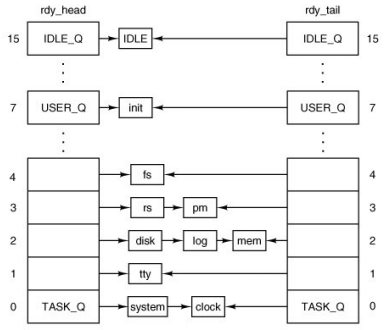
Minix guy

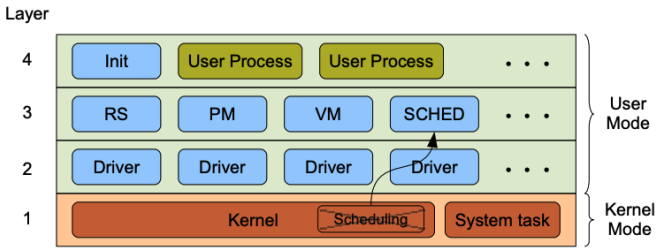
Linux guy

User Mode Scheduler

Move the scheduling mechanism from kernel mode to user mode.

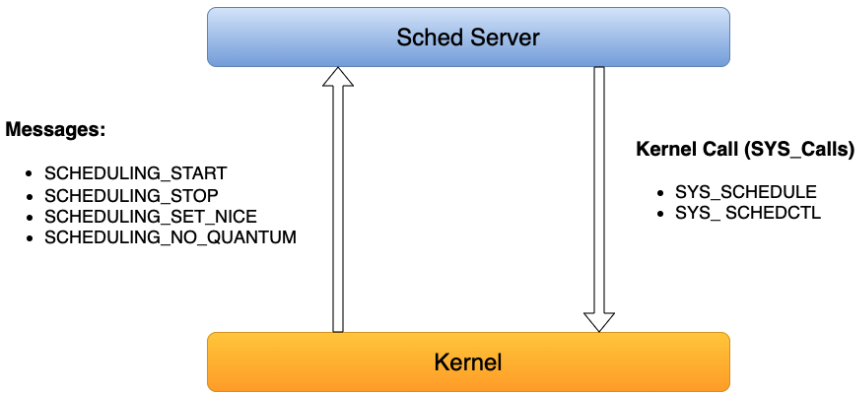
- Still uses MLFQ





How it works

When a process is created, stopped, nice system call is used, or a process is out of quantum, the sched server is called, and to actually change the the current queue two system calls are used.



Interface

- Two system library routines are exposed to user mode, **sys_schedctl** and **sys_schedule**.

These send messages to kernel requesting to start scheduling a particular process and to give a particular process quanta and priority, respectively.

Once a process runs out of its quantum, the user mode scheduler is notified with a message.

This message contains system and process feedback that the scheduler may use for making scheduling decisions.

Kernel Scheduling

- The Kernel information about the process in the process table such as:

Listing 1: Process table entries.

```
1  char p_priority;              /* current process priority */
2  u64_t p_cpu_time_left;        /* time left to use the cpu */
3  unsigned p_quantum_size_ms;   /* assigned time quantum in ms*/
4  struct proc *p_scheduler;     /* who gets out of quantum msg */
```

-

- Default Policy:
  - Although the scheduler is in user space, a simple Kernel policy still exists for failure and for the time before the sched server is up, that is when a process finishes its quantum it is assigned a new one and added to the end of the quantum.

Sched Server

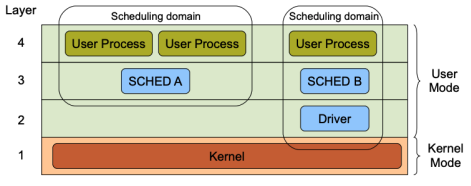
The current SCHED implementation mimics the policy that was previously implemented in kernel.

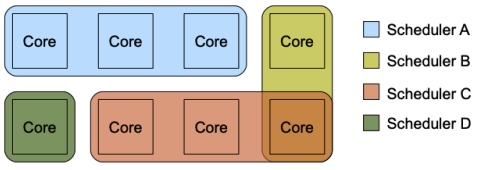
Each time a process runs out of quantum, it will be bumped down in priority by one.

Then, periodically, the scheduler will run through all processes that have been bumped down and push them up, one queue at a time.

Multiple Schedulers

Moving Scheduler into user space presents an important scheduling opportunity to create multiple schedulers, where a scheduler could exist per user, per device type, etc. Also, this allows better utilization of a multicore system as it allows higher cpu utilization and load balancing.





Minix IPC and Interrupt Handling

Minix IPC Message Types

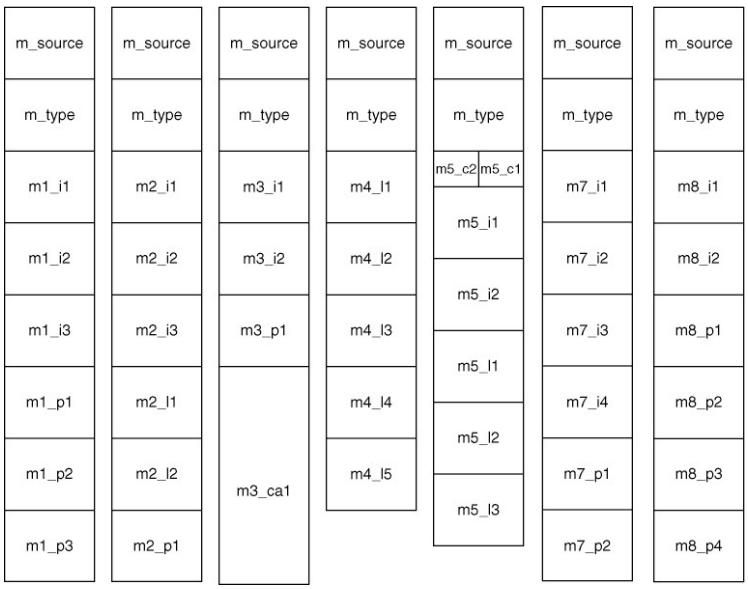
- 7 message types (see file **ipc.h**)
  - Defines *message* struct, and *send, receive, sendrecv*
- Message could have been array of bytes (unstructured)
  - Minix uses a union on message types
  - 7 types: message type 1 to 8 (6 is obsolete)
    - Message struct: m_source: who sent the message
      - » m_type: what is the message type
      - » data fields
  - Technically, x.m_u: union of struct, x.m_u.m_1.m1_i1
    - Use macros to simplify: x.m1_i1
    - data types: integer, long, pointer, char, char array

Minix IPC

- Use rendezvous communication (blocking)
  - send: check if destination waiting, copy msg to recv buffer,
    - otherwise block sender until receiver receives message
- **How are messages used?**
  - **Kernel calls**
  - **proc.c** sys_call
    - convert software interrupt into message, send and recv
    - actual work done in *mini_send, mini_recv, mini_notify*

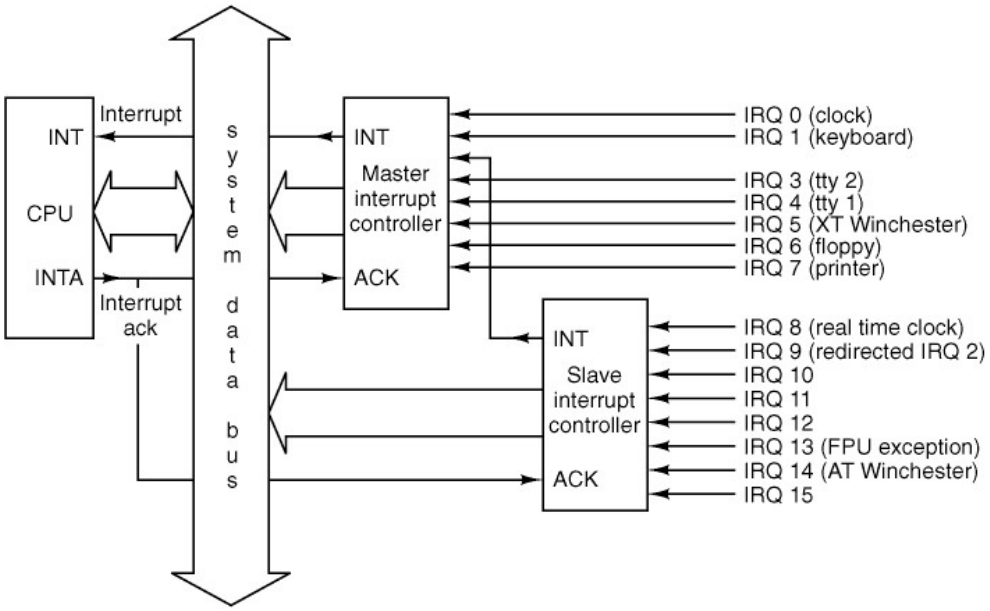
MINIX Message Types

The seven message types used in MINIX 3. The sizes of message elements will vary, depending upon the CPU architecture; figure shows 32-bit points for Intel



Interrupt Handling on Intel PCs

Interrupt processing hardware on a 32-bit Intel PC.



Interrupt Handling on Intel PCs

- Intel 32 bit CPUs: 2 controller chips
  - each handles 8 inputs: one master and one slave
  - 15 interrupts, IRQ 0: clock input
- IF INT raised, INT pin tells CPU that INT n occurred
- Interrupt vector: index a table of 256 entries
  - Minix uses 56 of these entries
  - Interrupt Gate Descriptors **(IGD)**
  - Jump to Interrupt Handler
  - CPU disables all other interrupts when an interrupt occurs.

Hardware dependent Interrupt Support

- Inteel 8259 controller chip: code in i8259.c
- Table stored on i8259 chip
  - Generates 8 bit index
  - CPU indexes into this table to find correct IGD entry
  - Jump to Interrupt handler
- Minix device drivers are in user space!
  - kernel holds a stub to interrupt handler
  - interrupt handling code is in user-space device driver
  - Up-call from kernel to user-space driver to handle interrupt

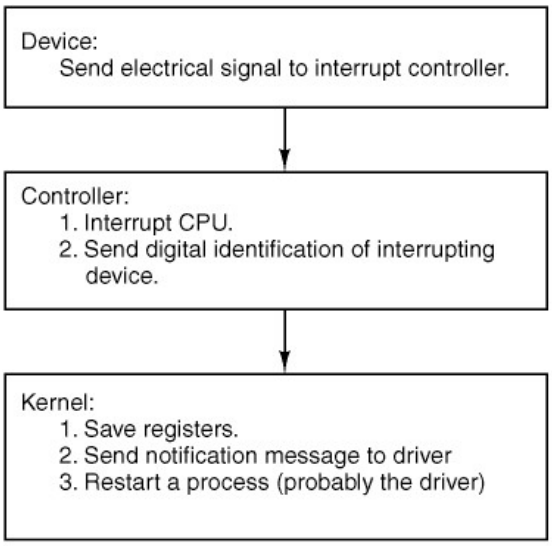
Minix Kernel Interrupt Handling

Skeleton of what the lowest level of the operating system does when an interrupt occurs.

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 marks waiting task as ready.
7. Scheduler decides which process is to run next.
8. C procedure returns to the assembly code.
9. Assembly language procedure starts up new current process.

Minix Kernel Interrupt Handling

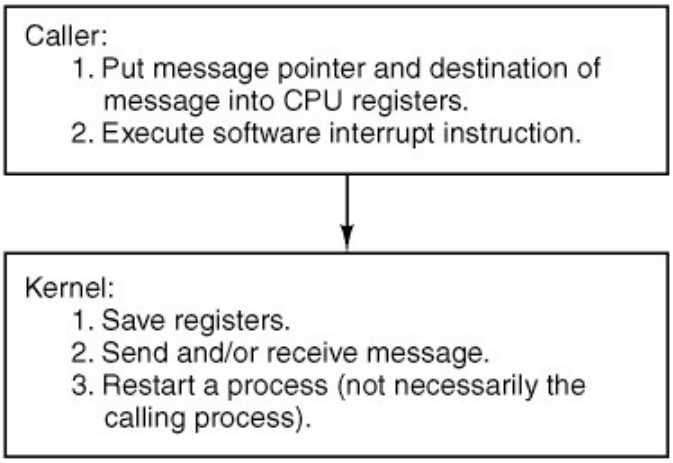
Figure 2-40. (a) How a hardware interrupt is processed.



(a)

Software Interrupts

How a system call is made.



(b)

Process Management

What is a process?

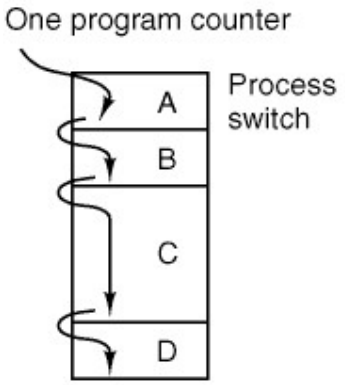
- The OS manages a variety of activities:
  - User programs
  - Batch jobs and command scripts
  - System programs: printers, spoolers, name servers, file servers, network listeners, etc.
- Each of these activities is encapsulated in a **process.**
- A process includes the execution context (PC, registers, VM, resources, etc.) and all the other information the activity needs to run.
- *A process is not a program*. A process is one instance of a program in execution. Many processes can be running the same program. Processes are independent entities.

What's in a Process?

- **Process: dynamic execution context of an executing program**
- Several processes may run the same program, but each is a distinct process with its own state (e.g., MS Word).
- A process executes sequentially, one instruction at a time
- **Process state** consists of at least:
  - the code for the running program,
  - the static data for the running program,
  - space for dynamic data (the heap), the heap pointer (HP),
  - the Program Counter (PC), indicating the next instruction,
  - an execution stack with the program's call chain (the stack), the stack pointer (SP)
  - values of CPU registers
  - a set of OS resources in use (e.g., open files)
  - process execution state (ready, running, etc.).

The Process Model (1)

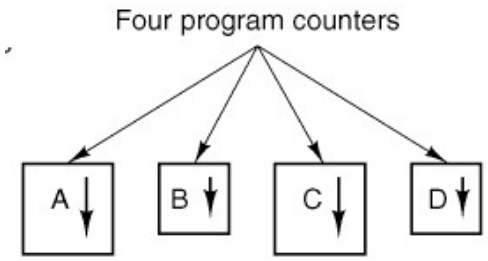
Multiprogramming of four programs.



(a)

The Process Model (2)

Conceptual model of four independent, sequential processes.



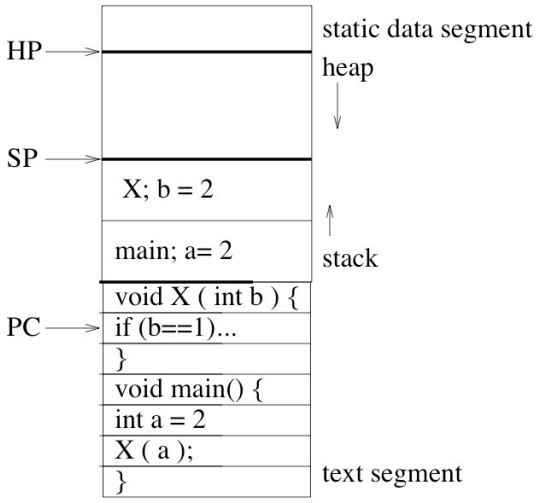
(b)

Example Process State in Memory

What you wrote:

```
  void X (int b){
PC ->  if ( b == 1 ) …
  }

  main(){
    int a = 2;
    X ( a );
  }
```

What's in memory



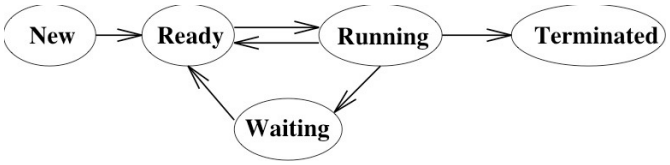
Process State

Process Execution State

- Execution state of a process indicates what it is doing

new:	the OS is setting up the process state
running:	executing instructions on the CPU
ready:	ready to run, but waiting for the CPU
waiting:	waiting for an event to complete
terminated:	the OS is destroying this process

- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).



Process Data Structures

- **Process Table :** OS data structure to keep track of all processes
  - The PCB tracks the execution state and location of each process
  - The OS allocates a new PCB on the creation of each process and places it on a state queue
  - The OS deallocates the PCB when the process terminates
- **The process table contains**:

- Process state (running, waiting, etc.)
- Process number
- Program Counter
- Stack Pointer
- General Purpose Registers
- Memory Management Information
- Username of owner
- List of open files
- Queue pointers for state queues
- Scheduling information (e.g., priority)
- I/O status
- …

Minix Process Table

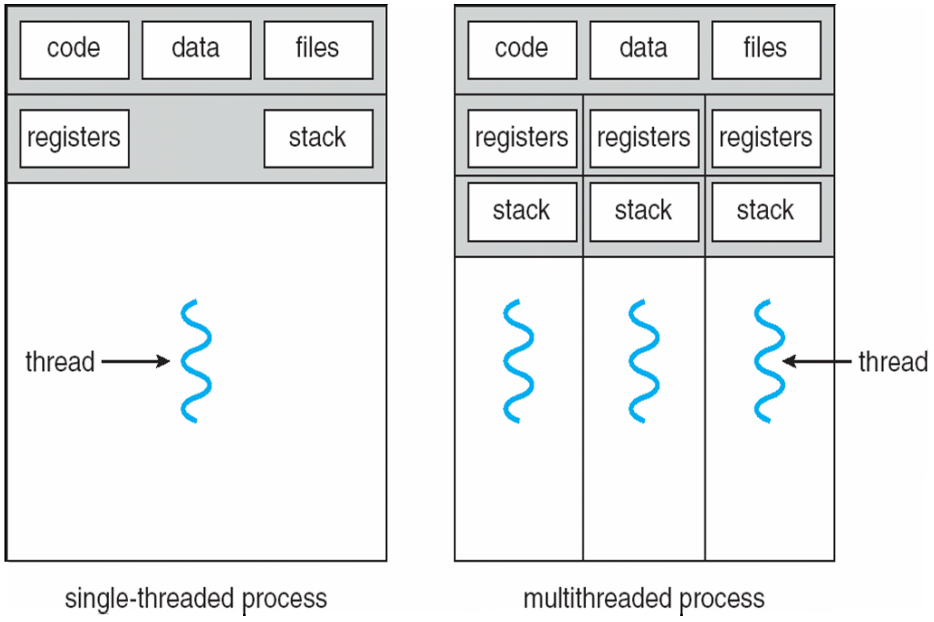
Fields of the MINIX 3 process table. The fields are distributed over the kernel, the process manager, and the file system.

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

Processes versus Threads

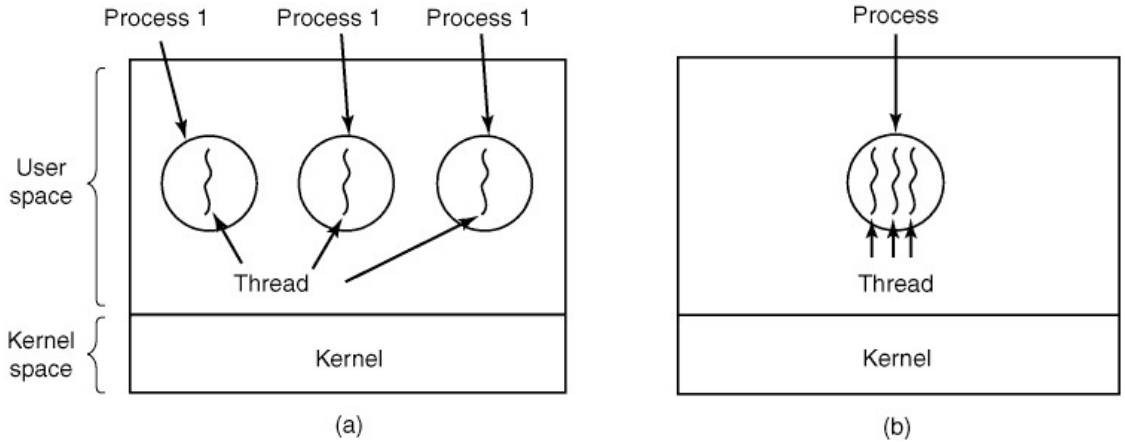
- **A process** defines the address space, text, resources, etc.,
- **A thread** defines a single sequential execution stream within a process (PC, stack, registers).
- Threads extract the *thread of control* information from the process
- Threads are bound to a single process.
- Each process may have multiple threads of control within it.
  - The address space of a process is shared among all its threads
  - No system calls are required to cooperate among threads
  - Simpler than message passing and shared-memory

Single and Multithreaded Processes



Threads vs Processes

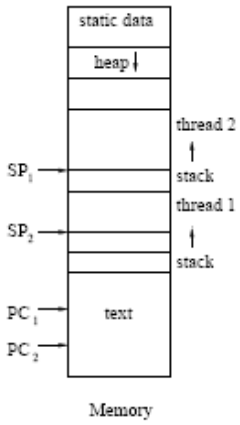
Three single-threaded processes vs one multi-threaded process



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).
- Note: The example has 3 threads: main, producer and consumer. The main thread exits after creating the producer and consumer and is *not shown.*

Threads

The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

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	

Kernel Threads

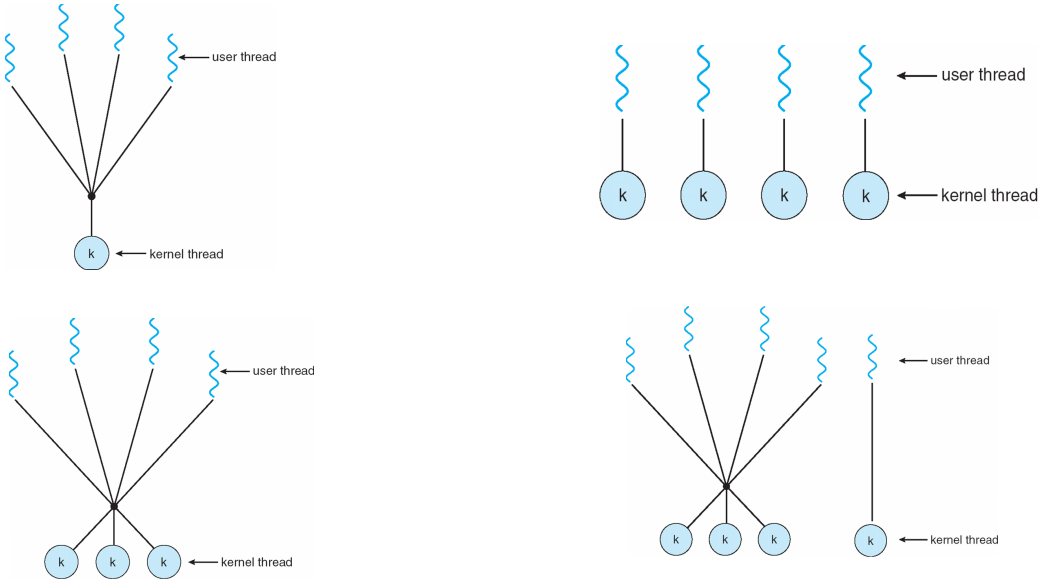
- **A kernel thread**, also known as a **lightweight process**, is a thread that the operating system knows about.
- Switching between kernel threads of the same process requires a small context switch.
  - The values of registers, program counter, and stack pointer must be changed.
  - Memory management information does not need to be changed since the threads share an address space.
- The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.

➔Switching between kernel threads is slightly faster than switching between processes.

User-Level Threads

- A **user-level thread** is a thread that the OS does *not* know about.
- The OS only knows about the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a *thread library* to manage threads (create and delete them, synchronize them, and schedule them).

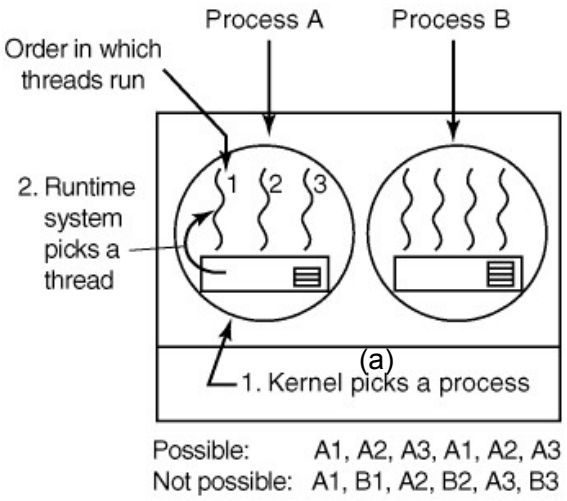
Threading Models



- Many-to-one, one-to-one, many-to-many and two-level

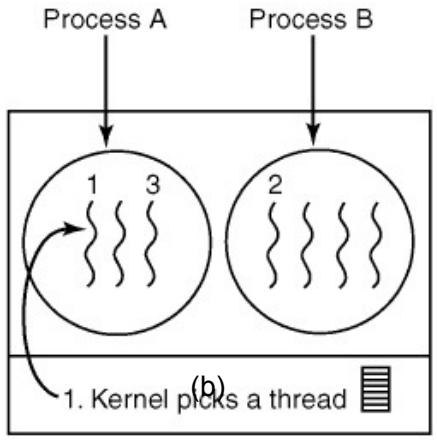
User-level Threads Scheduling

OS CPU scheduler picks a process, then run-time system picks a thread in that process



Kernel Thread Scheduling

OS CPU scheduler directly chooses a kernel thread (and hence the process) to schedule



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Pthreads and Win32 Threads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

- WIN32 Threads: Similar to Posix, but for Windows

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Examples

```
Pthreads:
    pthread_attr_init(&attr);  /* set default attributes */
    pthread_create(&tid, &attr, sum, &param);

Win32 threads
ThreadHandle  = CreateThread(NULL, 0, Sum, &Param, 0, &ThreadID);


Java Threads:

Sum sumObject = new Sum();
Thread t = new Thread(new Summation(param, SumObject));
t.start();  // start the thread
```

Thread Support in Minix

- Minix has no support for kernel-level threads
- User-level threads only choice
- **libmthread** - lightweight Minix user-level thread library
  - supports large subset of pthreads
- **GNU pthreads** - GNU pthreads library
  - full pthreads implementation as user-level threads

Process Creation

Process Creation

Principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Creating a Process

- One process can create other processes to do work.
  - The creator is called the *parent* and the new process is the *child*
  - The parent defines (or donates) resources and privileges to its children
  - A parent can either wait for the child to complete, or continue in parallel
- In Unix, the *fork* system call called is used to create child processes
  - Fork copies variables and registers from the parent to the child
  - The *only difference* between the child and the parent is the value returned by fork
    - In the parent process, fork returns the process id of the child
    - In the child process, the return value is 0
  - The parent can wait for the child to terminate by executing the *wait* system call or continue execution
  - The child often starts a new and different program within itself, via a call to *exec* system call.

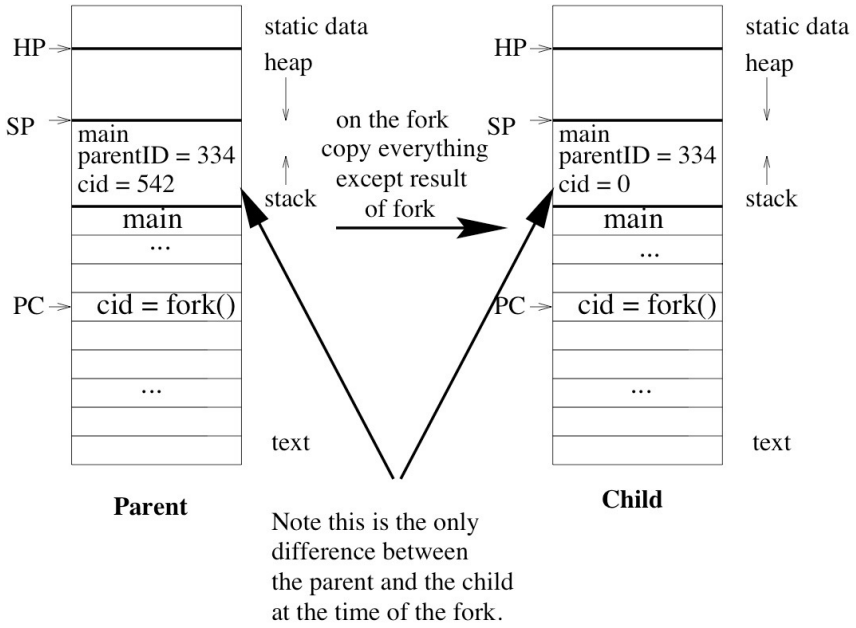
Creating a Process: Example

- When you log in to a machine running Unix, you create a shell process.
- Every command you type into the shell is a child of your shell process and is an implicit *fork* and *exec* pair.
- For example, you type emacs, the OS *"forks"* a new process and then *"exec"* (executes) emacs.
- If you type an & after the command, Unix will run the process in parallel with your shell, otherwise, your next shell command must wait until the first one completes.

Example Unix Program: Fork

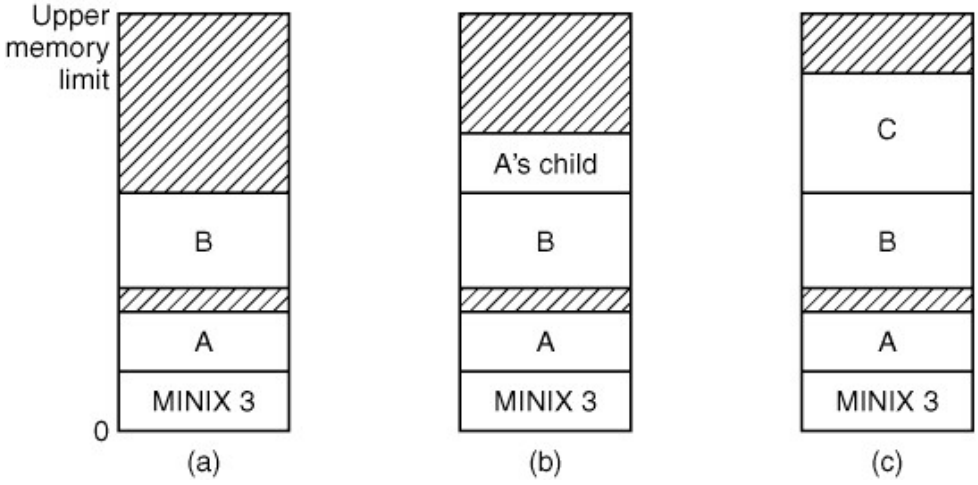
```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

main() {
  int parentID = getpid();    /* ID of this process */
  char  prgname[1024];
  gets(prgname); /* read the name of program we want to start */
  int cid = fork();
  if(cid == 0) {  /* I'm the child process */
    execlp( prgname, prgname, 0);  /* Load the program */
    /* If the program named prgname can be started, we never get
   to this line,  because the child program is replaced by prgname */
    printf("I didn't find program %s\n", prgname);
  } else { /* I'm the parent process */
    sleep (1);  /*  Give my child time to start. */
    waitpid(cid, 0, 0);  /*  Wait for my child to terminate. */
    printf("Program %s finished\n", prgname);
  }
}
```

What is happening on the Fork



Memory Layout for fork/exec

Figure 4-30. Memory allocation (a) Originally. (b) After a fork. (c) After the child does an exec. The shaded regions are unused memory. The process is a common I&D one.



Example Unix Program: Explanation

fork() forks a new child process that is a copy of the parent.

execlp() replaces the program of the current process with the named program.

sleep() suspends execution for at least the specified time.

waitpid() waits for the named process to finish execution.

gets() reads a line from a file.

Minix FORK System Call

1. Check to see if process table is full.
2. Try to allocate memory for the child's data and stack.
3. Copy the parent's data and stack to the child's memory.
4. Find a free process slot and copy parent's slot to it.
5. Enter child's memory map in process table.
6. Choose a PID for the child.
7. Tell kernel and file system about child.
8. Report child's memory map to kernel.
9. Send reply messages to parent and child.

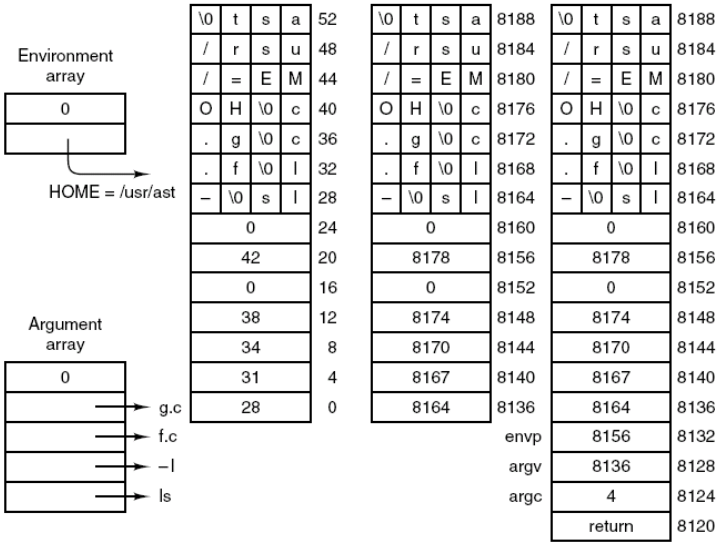
EXEC System Call in Minix

The steps required to carry out the exec system call.

1. Check permissions—is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle setuid, setgid bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

EXEC System Call (2)

(a) The arrays passed to *execve*. (b) The stack built by *execve*. (c) The stack after relocation by the PM. (d) The stack as it appears to *main* at start of execution.



EXEC System Call (3)

The key part of *crtso*, the C run-time, start-off routine.

```
push  ecx          ! push environ
push  edx          ! push argv
push  eax          ! push argc
call  _main        ! main(argc, argv, envp)
push  eax          ! push exit status
call  _exit
hlt                ! force a trap if exit fails
```


Process Termination

Conditions that cause a process to terminate:

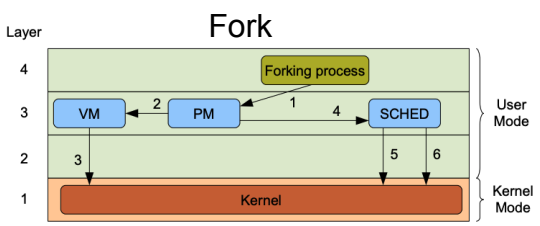
1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

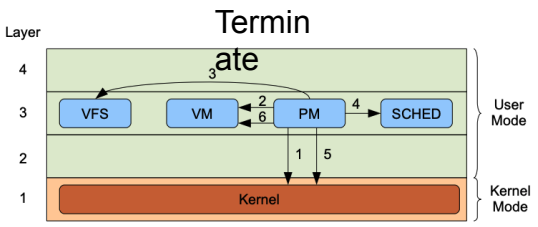
Process Termination

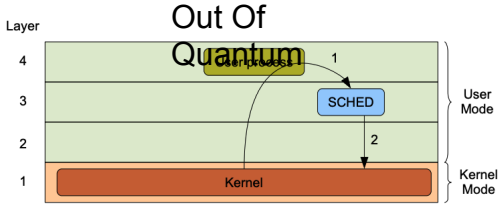
- On process termination, the OS reclaims all resources assigned to the process.

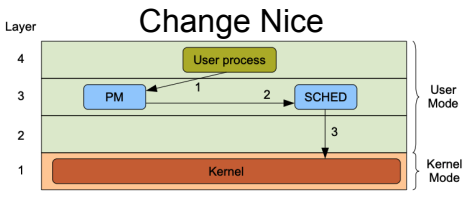
- In Unix
  - a process can terminate itself using the *exit* system call.
  - a process can terminate a child using the *kill* system

Message Flow









Performance

The performance of this approach is related to the performance overhead by minix itself, due to the IPC Mechanism used. So in the following cases the performance is measured using in context switching cased by IPCs.

Fork: To execute a fork it requires (6 mode switches and 2 context switches).

$$\begin{array}{ccccccccccc} \cdots & \longrightarrow & PM & \xrightarrow[1]{mode} & kernel & \xrightarrow[1]{cxt} & SCHED & & & & \\ & \xrightarrow[2]{mode} & kernel & \xrightarrow[3]{mode} & SCHED & \xrightarrow[4]{mode} & kernel & & & & \\ & \xrightarrow[5]{mode} & SCHED & \xrightarrow[6]{mode} & kernel & \xrightarrow[2]{cxt} & PM & \longrightarrow & \cdots & & \end{array}$$

Termination: 2 mode switches, 2 context switches

$$\begin{array}{ccccccc} \cdots & \longrightarrow & PM & \xrightarrow[1]{mode} & kernel & \xrightarrow[1]{cxt} & SCHED \\ & \xrightarrow[2]{mode} & kernel & \xrightarrow[2]{cxt} & PM & \longrightarrow & \cdots \end{array}$$

Out of Quantum: 1 mode switch, 1 context switch

$$kernel \xrightarrow[1]{cxt} SCHED \xrightarrow[1]{mode} kernel$$