Chapter 3
Process Description and Control

Req. of an OS wrt. Processes

- Use multiple processes to maximize processor utilization while providing reasonable response time
  - Uniprocessor: interleave execution of procs.
  - Multiprocessor: interleaving and parallel execution of procs.
- Allocate resources to processes
- Support interprocess communication and user creation of processes

Previously Introduced Concepts

- Computer platform consists of a collection of hardware resources
  - Processor, main memory, timers, IOs
- Computer applications are developed to perform some task
  - Bespoke system vs general workstations
- Inefficient for applications to be written directly for a given hardware platform
  - Code reuse among applications
  - Need to manage multiprogramming (do you really want to put this into the application?)
  - Protect data, IO, etc (can you put this into the application?)
- Some still believe otherwise
  - Consider porting assembly code?

Previously Introduced Concepts

- OS provides a convenient, feature rich, secure, and consistent interface for applications to use

- OS provides a uniform, abstract representation of resources that can be requested and accessed by application => a virtual machine

HOW DOES THE OS MANAGE PROCESSES?

Process (possible definitions)

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by
  - execution of a sequence of instructions
  - current state
  - associated set of system instructions

Process (revisited)

- Consists of three components
  - An executable program
  - Associated data needed by the program
  - **Execution context of the program**
    - All information the operating system needs to manage the process
- So what are the process elements in detail?

Process Elements

- Identifier
  - Usually abbreviated as PID
- State
  - E.g., running state
- Priority
  - relative to other processes

Process Elements

- Memory pointers
  - Pointers + shared memory blocks
- Context data
  - Registers, PSW, program counter
- I/O status information
  - Outstanding I/O requests, used I/O devices
- Accounting information
  - Amount of processor time, time limits, threads
  - Try out ‘top’, ‘ulimit’

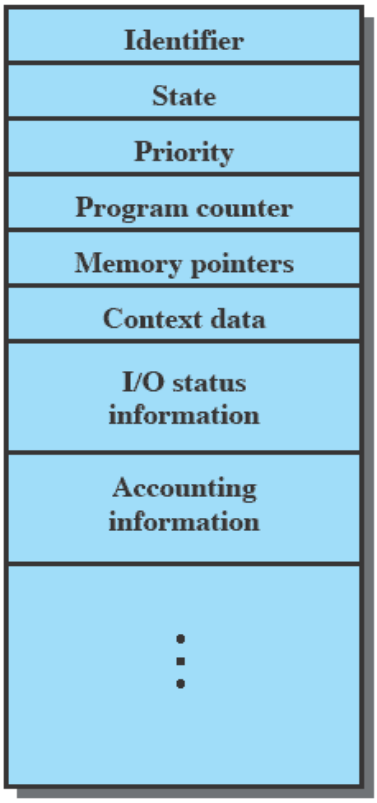
Process Control Block (PCB)

- The data structure that contains the process elements
- Created and managed by the operating system
- Allows support for multiple processes

Question!

Do you allow the user to manipulate the elements of the process control block?
Not directly, partially yes (e.g., set priority through command)

Process Control Block



Varies between OSs
=> Very small for embedded OS

Figure 3.1 Simplified Process Control Block

Example PICOS18

```
typedef rom const struct _rom_desc_tsk
{
   unsigned char prioinit;
   unsigned char *stackAddr;
   void (*addr_ROM)(void);
   unsigned char tskstate;
   unsigned char tskid;
   unsigned int  stksize;
} rom_desc_tsk;
```

Instantiation

```
/*****************************************************************
 * ---------------------------- task VM ----------------------------
 *****************************************************************/
rom_desc_tsk rom_desc_task_vm = {
    TASK_VM_PRIO,                   /* prioinit from 0 to 15       */
    _stack_vm,                     /* stack address (16 bits)     */
    TASK_VM,                       /* start function     */
    READY,                        /* state at init phase         */
    TASK_VM_ID,                     /* id_tsk from 0 to 15         */
    sizeof(_stack_vm)                /* stack size    (16 bits)     */
};
```

Trace of the Process

- Sequence of instruction that execute for a process
- Dispatcher switches the processor from one process to another

The trace by the OS is not necissary a sequence of instructions (that is how it looks to the process though)

Example Execution (Setup)

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

Example Execution (Memory Layout)

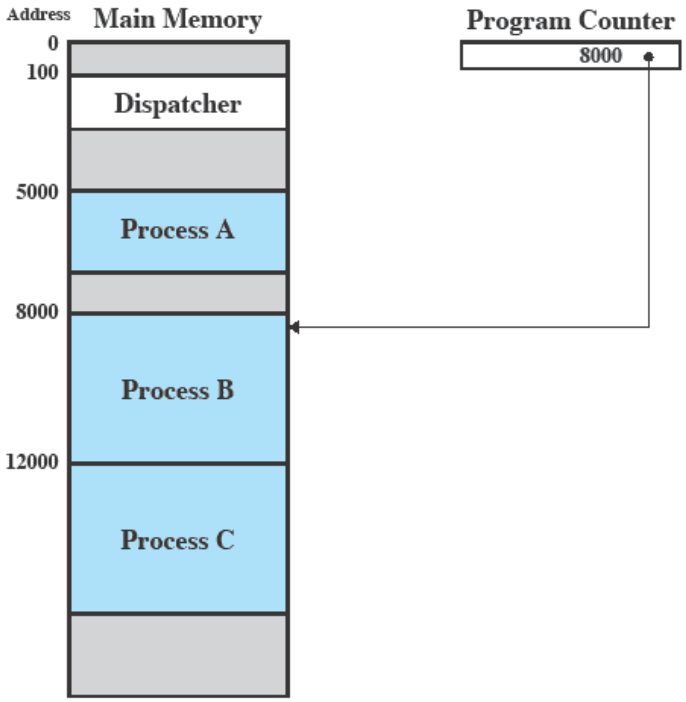


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

Combined Trace of All Processes

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------	Timeout
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----------	I/O Request
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------	Timeout
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------	Timeout
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------	Timeout

100 = Starting address of dispatcher program

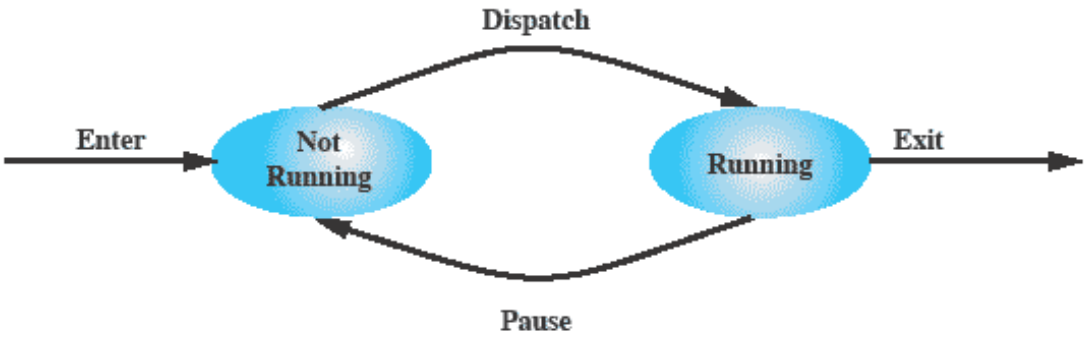
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

We can see here that to the OS a trace is not sequential because there are breaks where we make IO requests and have timeouts. We dont actually have processes executing in parallel, but the processor just jumps between them alot.

Two-State Process Model

- Looking at the previous example: A process may be in one of two states
  - Running
  - Not-running



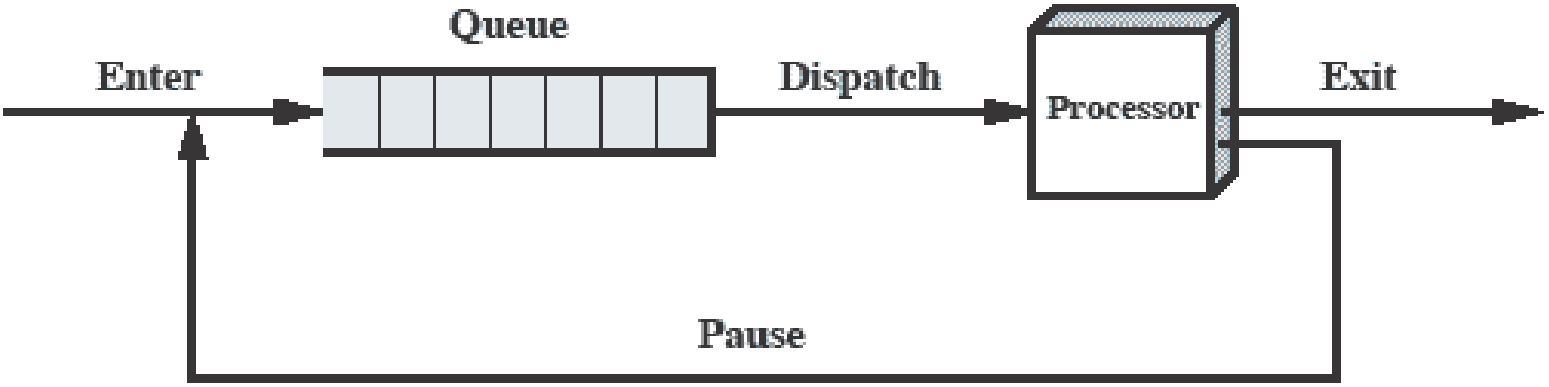
(a) State transition diagram

- How to accommodate model in OS design?
  - We already have PCBs and need …

This is a very simple model to hold the state of the process. Either the proccess is running or not. The process just gets looped between those states every time interrupts happen. We have a pointer to the currently running process (remember that process have their state saved on them). When an interrupt happens the dispatcher grabs the next processor that is not running to resume work on that.

OS Structure (so far)

1. Store data about the process (process control block)
2. Processes must wait in some sort of queue until it's their turn.



(b) Queuing diagram

When a process is not running on the cpu we put in on the queue and switch its state to running and pop it off when the cpu is free

Process Creation

Table 3.1 Reasons for Process Creation

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process spawning: a parent process explicitly creates a child process.
→ check this with 'ps'

Process Termination

Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded **ulimit**	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.

Explain: core dump

A core dump is the current execution context at the process termination.

Process Termination

Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Explain: 'kill –SIGTERM' and others (compare SIGKILL and SIGTERM)

Simple Queuing Mechanism is Inefficient

- ... because some processes are
  - Not-running but ready to execute
  - Not-running and blocked

- With a single queue: Dispatcher must scan list to find process not-running, ready, and in queue the longest
- With multiple queues: first pick the right queue and then go round robin.

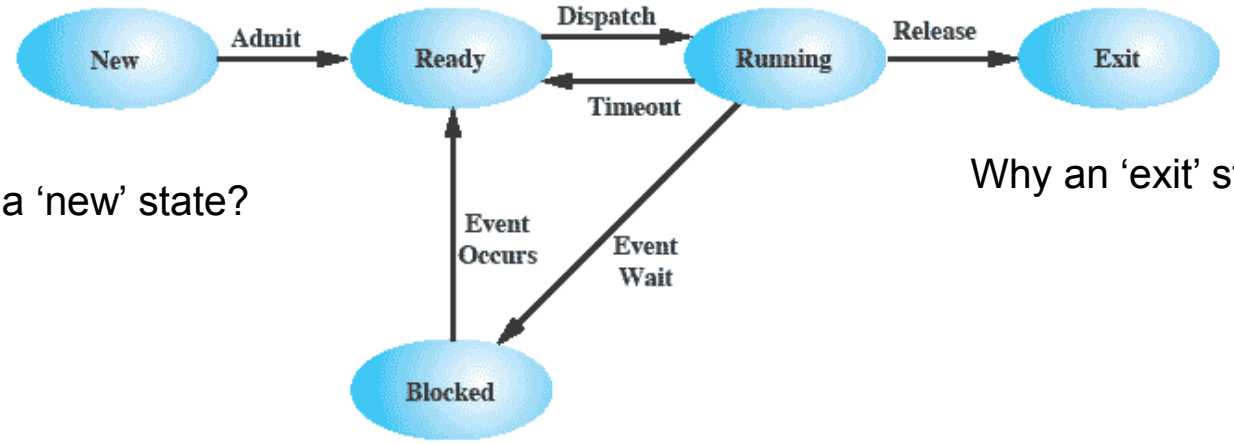
But what multiple queues should we have and how does it affect the states?

A Five-State Model

- Refine the not-running state:

- **Running:** the process currently executed
- **Ready:** a process that can be executed
- **Blocked/Waiting:** a process that cannot execute, because it waits for something
- **New:** a new process to enter the system
- **Exit:** a halted or aborted process

Five-State Process Model



Why a ‘new’ state?

Why an ‘exit’ state?

Figure 3.6 Five-State Process Model

Explain: preemption

- Ready = the process has been set up
- Running = the process is currently running
- Blocked = the process is waiting for something (like an io read)
- New = The memory is not yet fully in memory
- Exit = The clean up after process has terminated

This state manchine is managed by the dispatcher. We will need a more complicated queue system to manage all of this.

Process States

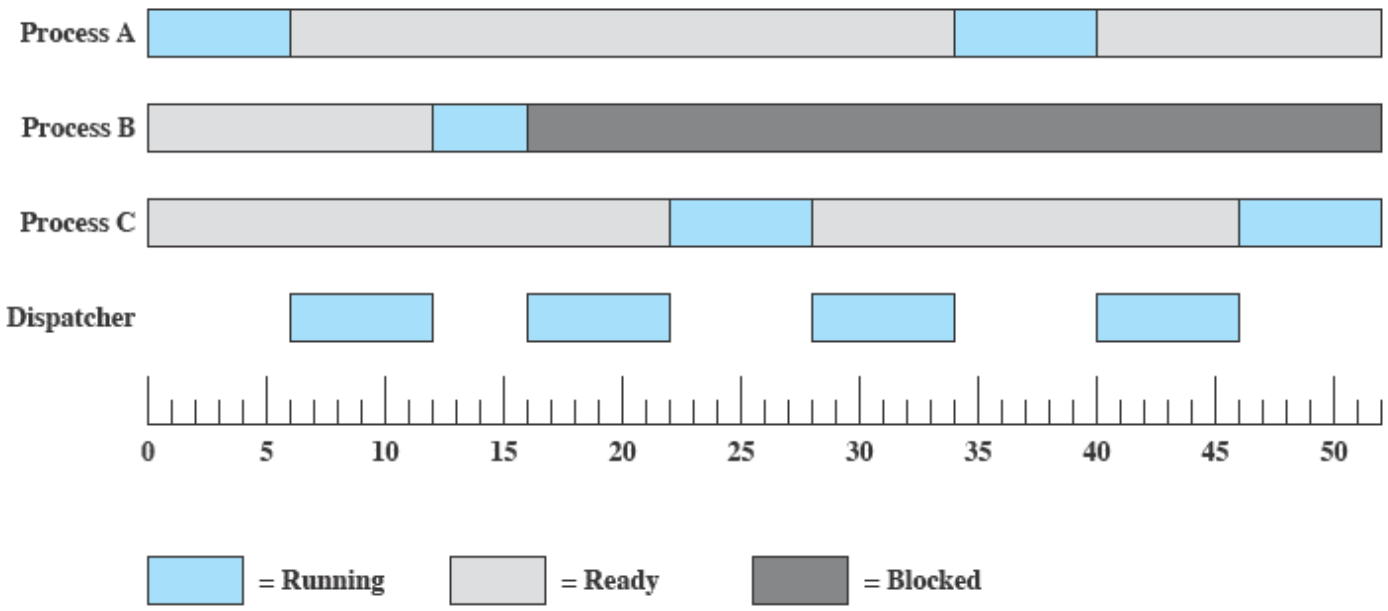
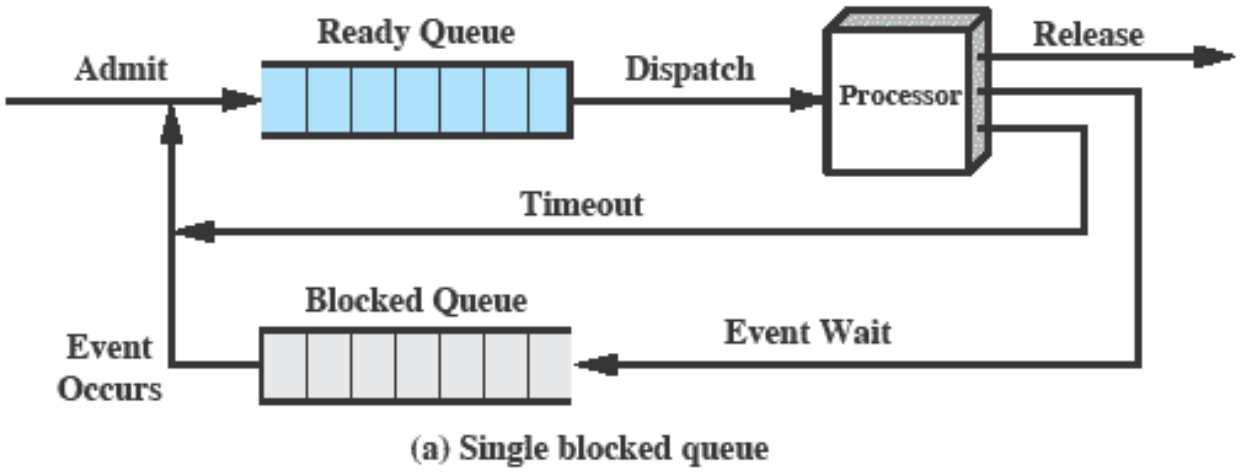


Figure 3.7 Process States for Trace of Figure 3.4

Using Two Queues



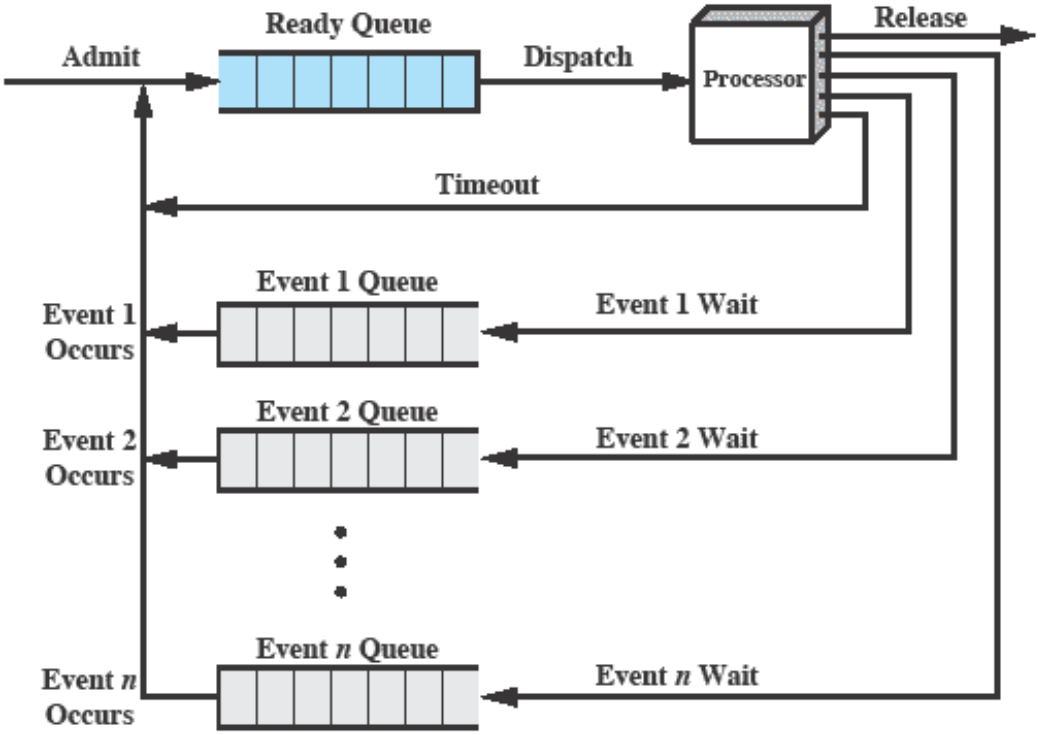
(a) Single blocked queue

What's the problem with it?

Problem: unblocking a task requires the OS to search through that list

The ready queue has all of the processes ready to run (ie. of that state). Then the dispatcher executes for reasons stated above, it grabs the next ready process. When that process leaves the running state the processor is released and starts again. If the process is blocked it goes onto a blocked queue which then goes back to be admited and pushed onto the bottom of the ready queue. This also allows for timeouts to kick a stalled process back onto the wait queue to free the processor. The problem with this design is that there will be many IO's.

Multiple Blocked Queues



(b) Multiple blocked queues

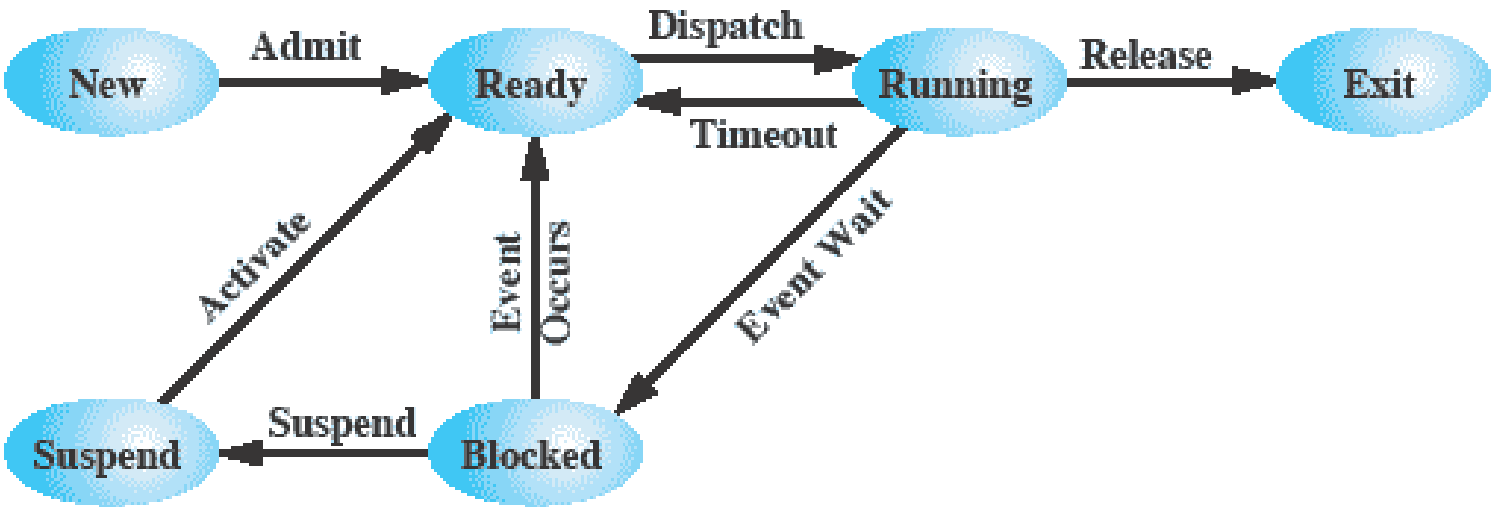
To deal with inefficiencies of multiple of IO's we make a queue for each IO. Every time a specific event occurs it the processor puts it onto that specific queues to wait there and just pop off the top one when its done. Now we dont have to look for a specific process when it's specific ready call is done. So if two different blocks happen they go onto two different queues that each wait for their own unblock signal when previously we would have to search through the block queue for the correct process.

Suspended Processes

- Problem: Processor is faster than I/O so all processes could be waiting for I/O
- Solution: admit more processes
- Swap blocked processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk
- Two new states
  - Blocked/Suspend
  - Ready/Suspend

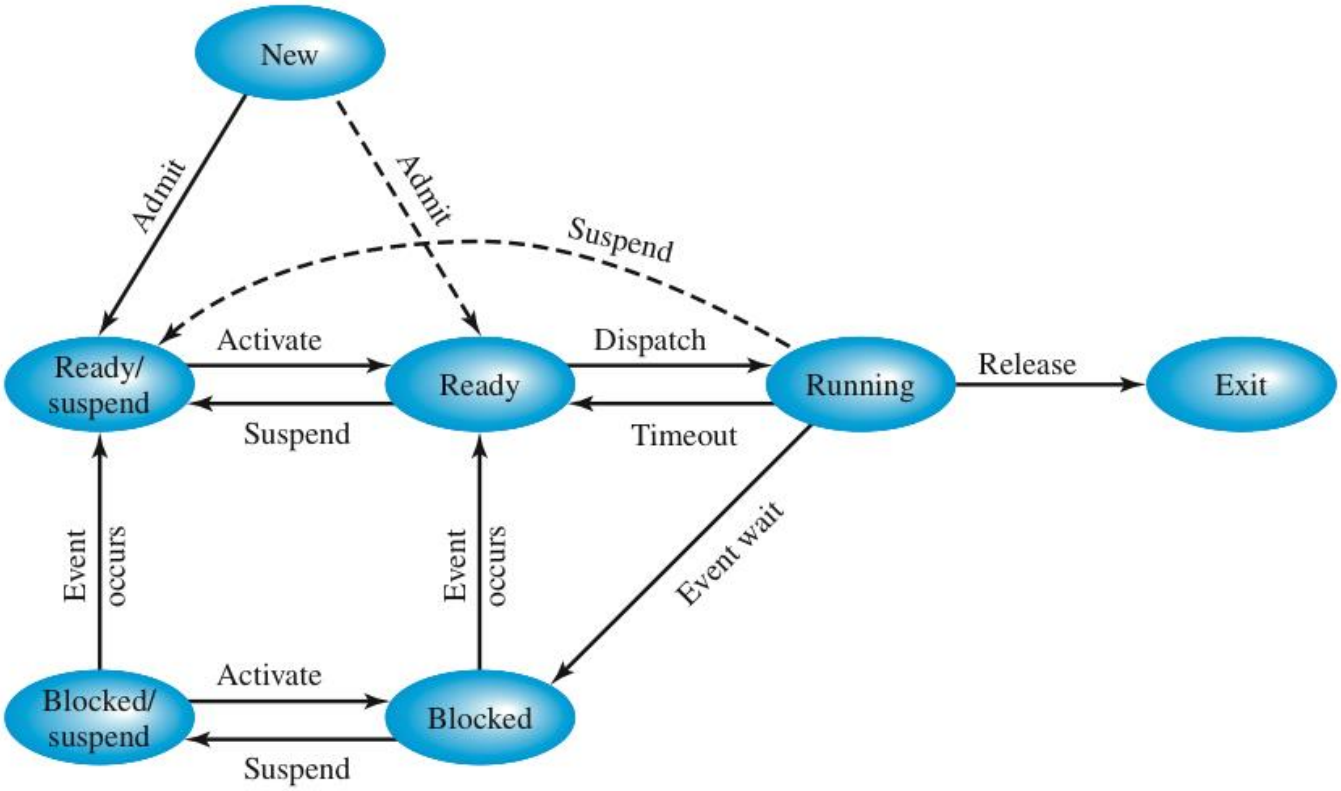
We added a suspended state to allow us to admit more processes. This is when the program is paused to free up more memory on the stack. For a program to be running it needs to be loaded into main memory.

One Suspend State



(a) With One Suspend State

Two Suspend States



If we have memory available it goes to the ready state, else it goes to the suspend state and none of the necissary stuff is loaded into memory. When there is enough space we put it into the ready queue to be executed like normal. We have a similar queue when a blocked process tries to enter the ready queue and cannot. This lets us admit as many processors as possible into our system without memory being a constraint.

Reason for Process Suspension

Table 3.3 Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Notes!

HOW DOES THE OS MANAGE RESOURCES?

Processes and Resources

Dashed line = Waiting

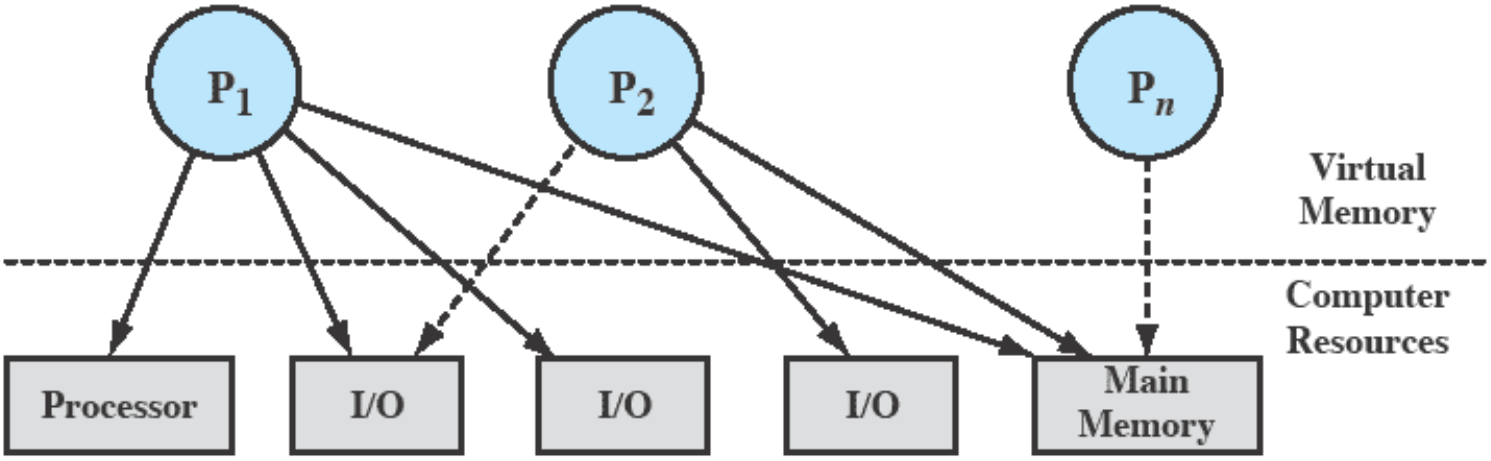


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

A process is assigned to one or more processors and might use peripherals during its execution. These all need to be managed by the OS.

Operating System Control Structures

- Information about the current status of each process and resource
- Tables are constructed for each entity that the operating system manages

We manage things by storing them in tables with maps.

Memory Tables

- … used to keep track of main & secondary (virtual) memory
- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

Overtime main memory becomes fragmented which we need to keep track of and what secondary storage is allocated to processes. Ditto for virtual memory.

I/O Tables

- … used by the OS to manage I/O devices (see /proc directory)
- I/O device is available or assigned
- Status of I/O operation
- Location in main memory being used as the source or destination of the I/O transfer
- Example: /proc directory

We use IO tables to track which IOs are in use and what memory they are using.

File Tables

- Existence of files
- Location on secondary memory
- Current Status
- Attributes (e.g., rwxr--r--)
- Sometimes this information is maintained by a file management system
- Example: fuser –u /

What files are allocated, their location, protection attributes, and what process they are locked to.

Process Table

- Where process is located in memory
- Attributes in the process image
  - Program
  - Data
  - Stack
  - Process control block (aka task control block, process descriptor, task descriptor)

  - Question: How/when does the OS create these tables?

A process table stores the id, where the process is in memory, its attributes, what data it uses, its stack start, and the process control block. All of the above listed tables are created during boot time and saved at shut down.

Process Image

Table 3.4 Typical Elements of a Process Image

User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program

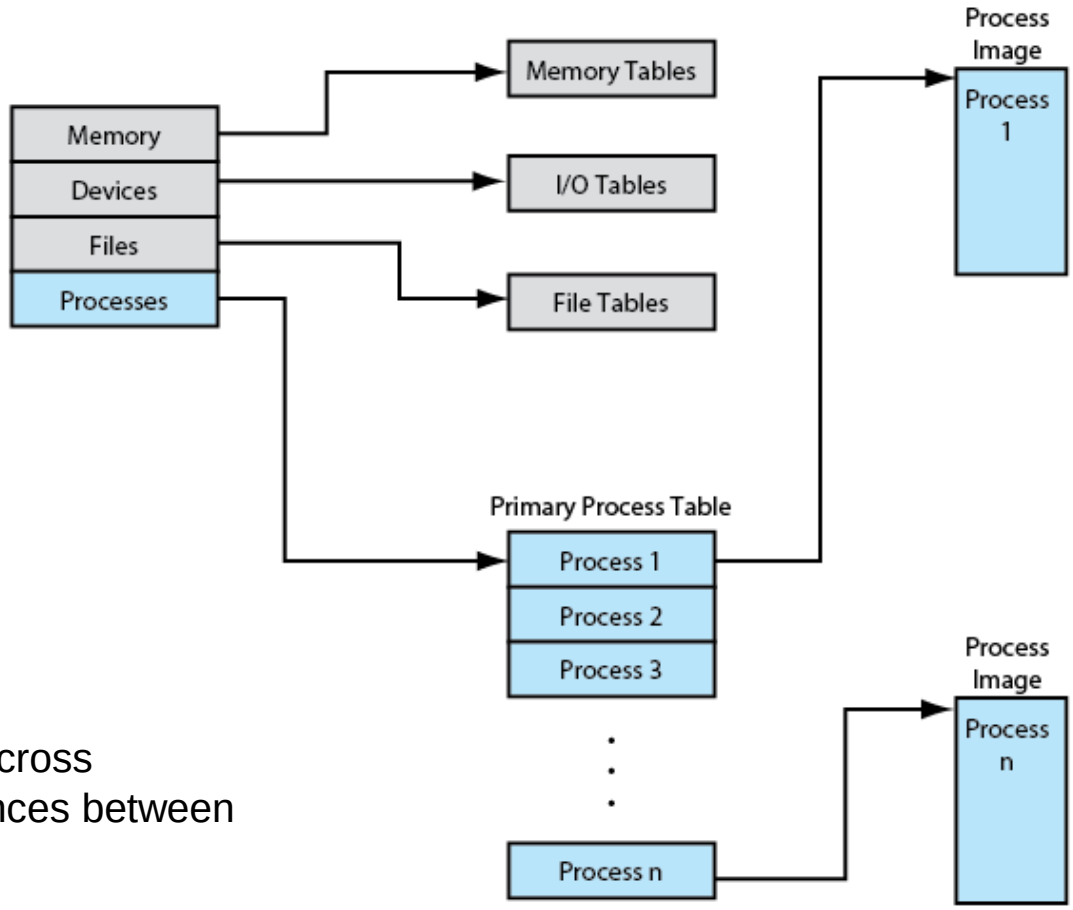
The program to be executed.

System Stack

Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

Process Control Block

Data needed by the operating system to control the process (see Table 3.5).



Lacks cross references between tables.

Figure 3.11 General Structure of Operating System Control Tables

This is a overview of the process table implementation. For each of the resources we have a table which are initialized at boot. For processes the table contains links to the process image.

Process Control Block

- Process identification
  - Identifiers
    - Numeric identifiers that may be stored with the process control block include
      - Identifier of this process (= unique key)
      - Identifier of the process that created this process (parent process)
      - User identifier
      - Defined by pid_t
        (demo: chase down pid_t on ece.uwaterloo.ca)

Process Identification(PID): when you have processes that spawn processes you need to keep track of what their parents are. Processes that are started during boot time are mapped to process one.

Process Control Block

- Processor State Information
  - User-Visible Registers
    - A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.
    - Might be as low as only 1 working register

Process Control Block

- Processor State Information
  - Control and Status Registers

    These are a variety of processor registers that are employed to control the operation of the processor. These include
    - *Program counter:* Contains the address of the next instruction to be fetched
    - *Condition codes:* Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
    - *Status information:* Includes interrupt enabled/disabled flags, execution mode

Process Control Block

- Processor State Information
  - Stack Pointers
    - Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Process Control Block

- Process Control Information
  (… meta info for handling processes, differs for each OS → design criteria)
  - Scheduling and State Information
    - Needed to perform scheduling function:
    - Process state: e.g., running, ready, waiting, …
    - Priority: e.g., 0-255
    - Scheduling related information: e.g., cpu time, slack, deadlines
    - **Importance**

Process Control Block

- Process Control Information
  - Data Structuring
    - Linked list with for child processes
    - Linked list for same priority processes
    - Linked list for ‘cohort’ processes

Process Control Block

- Process Control Information
  - Interprocess Communication
    - Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.
  - Process Privileges
    - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Process Control Block

- Process Control Information
  - Memory Management
    - This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
  - Resource Ownership and Utilization
    - Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Processor State Information

- Contents of processor registers
  - User-visible registers
  - Control and status registers
  - Stack pointers
- Program status word (PSW)
  - contains status information
  - Example: the EFLAGS register on Pentium machines

Modes of Execution

- User mode
  - Less-privileged mode
  - User programs typically execute in this mode
- System mode, control mode, or kernel mode
  - More-privileged mode
  - Kernel of the operating system

- Can you think of why you want more modes?
- → granular access (crappy drivers can't kill the machine), virtual machines, sandboxing downloaded programs

Steps in Process Creation

1. Assign a unique process identifier
2. Allocate space for the process
3. Initialize process control block
4. Set up appropriate linkages
   - Ex: add new process to linked list used for scheduling queue
5. Create of expand other data structures
   - Ex: maintain an accounting file

When to Switch a Process

- Looks easy, but is quite tricky if you want to provide guarantees (e.g., bandwidth, performance, etc)

Possible choices when to switch:

- Clock interrupt
  - process has executed for the maximum allowable time slice
- I/O interrupt (= I/O completed)
- Memory fault
  - memory address is in virtual memory so it must be brought into main memory

When to Switch a Process

- Trap
  - error or exception occurred
  - may cause process to be moved to Exit state
  - Used for debugging (ICD)
- Supervisor call
  - Switch to kernel process

Steps for a Process Switch

- Save context of processor including program counter and other registers
- Update the process control block of the process that is currently in the Running state
- Move process control block to appropriate queue – ready; blocked; ready/suspend
- Select another process for execution

Change of Process State

- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of the selected process

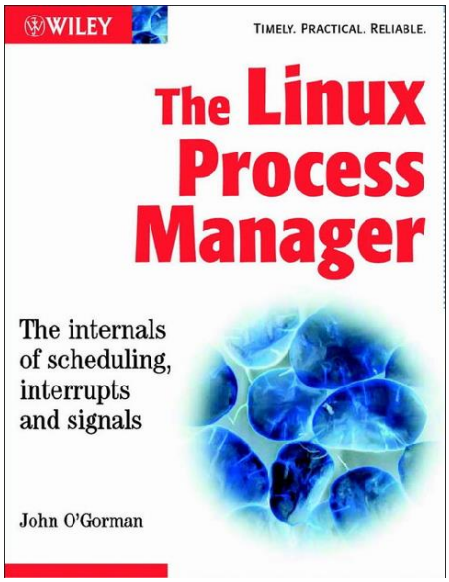
References

Recommended Reading:

- Chapter three of Stallings book

<u>Optional</u> Reading:

- O'Gorman, "The Linux Process Manager – The internals of scheduling, interrupts and signals", Wiley 2003 (Ch 1-3)



PROCESS SWITCH REVISITED

In part based on slides by S. Fischmeister, A. Tanenbaum & Mark Handley

Review: When to Switch Processes

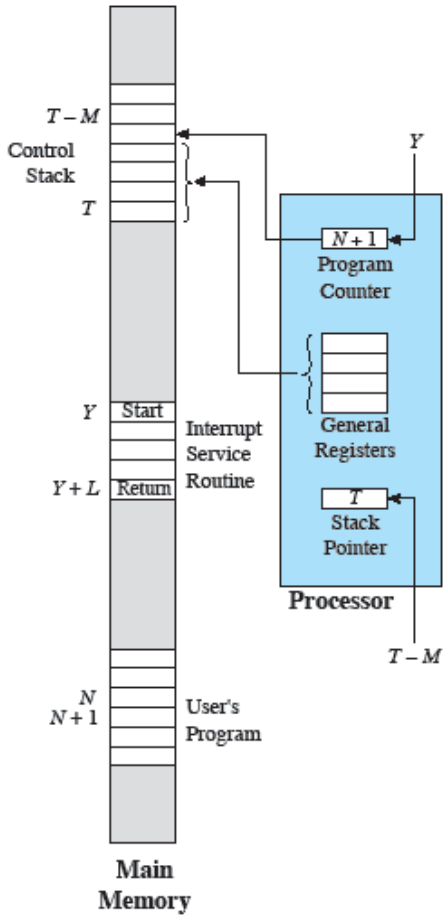
- **Clock interrupt**
- **I/O interrupt (= I/O completed)**
- **Memory fault**
- **Trap**
- **Supervisor call**

Review: Process Switching Steps

1. Save context of processor including program counter and other registers
2. Update the process control block of the process that is currently in the Running state
3. Move process control block to appropriate queue – ready; blocked; ready/suspend
4. Select another process for execution
5. Update the process control block of the process selected
6. Update memory-management data structures
7. Restore context of the selected process

An Context Switch Occurs

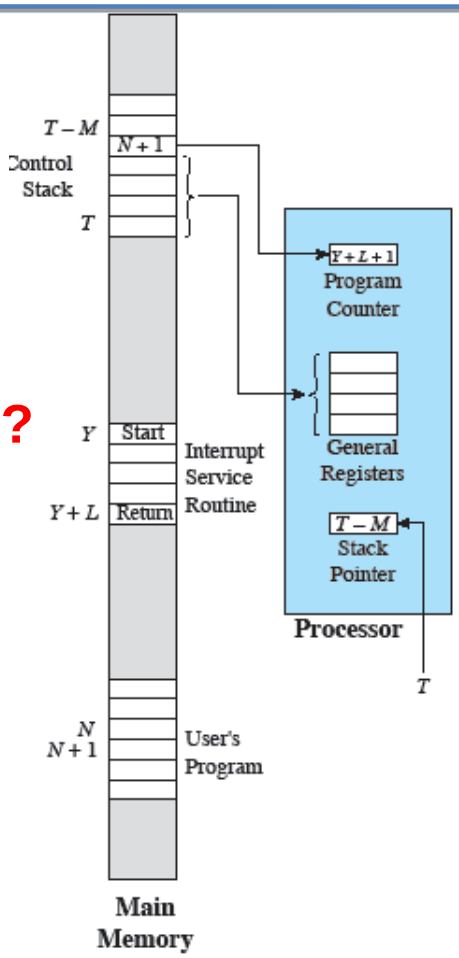
Storing a snapshot.



(a) Interrupt occurs after instruction at location *N*

Resuming Another Process

Restoring from a snapshot.



(b) Return from interrupt

Does it look familiar?

Example: PICOS18

- Kernel for Microchip PIC18 controllers
- Many functions of a microkernel
  - Events
  - Interrupts
  - Counters/alarms
  - Multitasking

Schedule()

```
/**************************************************************************
 * Force a scheduler action
 *
 * @return Status    E_OK if a service is called inside an ISR
 *                   or never returns
 *************************************************************************/
StatusType Schedule(void)
{
  INTCONbits.GIEL = 0;
  kernelState |= SERVICES;
  if (kernelState & ISR)
    return (E_OK);
  kernelState &= ~SERVICES;
  if (kernelState & USER)
    SAVE_TASK_CTX(stack_low, stack_high);
  SCHEDULE;
  return (E_OK);
}
```

OS does not use HW protection, instead "kernel mode" is emulated.

Step 1: Save all context info

Step 2-7: …

SAVE_TASK_CTX() – Part 1

```
#define SAVE_TASK_CTX(stack_low, stack_high)                    \
{                                                               \
   /* Disable global interrupt. */
_asm                                                            \
      bcf    INTCON, 6, 0                                       \
      movff  STATUS, PREINC1
      movff  WREG, PREINC1
_endasm                                                          \
   /* Store the necessary registers to the stack. */            \
_asm                                                            \
      movff  BSR, PREINC1
      movff  FSR2L, PREINC1
      movff  FSR2H, PREINC1
      movff  FSR0L, PREINC1
      movff  FSR0H, PREINC1
      movff  TBLPTRU, PREINC1
      movff  TBLPTRH, PREINC1
      movff  TBLPTRL, PREINC1
      movff  TABLAT, PREINC1
      movff  PRODH, PREINC1
      movff  PRODL, PREINC1
   _endasm                                                      \
```

Hint: Disable means hold interrupts.

Pending interrupts serviced, when enabled again

WREG – working register
FSR – address registers
TBDLPTRx, TABx – other address registers

Hint: PIC supports several indirect addressing modes through address registers.

Coldfire is “easier”

SAVE_TASK_CTX() – Part 2

```
/* Store the .tempdata and MATH_DATA areas. */                    \
  _asm
     movlw   TEMP_SIZE+1
     clrf    FSR0L, 0
     clrf    FSR0H, 0
  _endasm
     while (WREG--)
     {
       _asm
         movff   POSTINC0, PREINC1
       _endasm
     }
```

Hint:
tmpdata = temporary variables (used by ISRs).

MATH_DATA = return values used from mathematical library Functions (compiler extensions)

All data from embedded libs that assume a single program need to be stored in the context as well.

SAVE_TASK_CTX() – Part 3

```
/* Store the HW stack area. */                          \
    _asm                                                \
      movff   STKPTR, FSR0L                             \
    _endasm                                             \
      while (STKPTR > 0)                                \
      {                                                 \
        _asm                                            \
          movff   TOSL, PREINC1                         \
          movff   TOSH, PREINC1                         \
          movff   TOSU, PREINC1                         \
          pop                                           \
        _endasm                                         \
      }                                                 \
```

SAVE_TASK_CTX() – Part 4

```
 /* Store the number of addresses on the HW stack */            \
   _asm                                                         \
      movff   FSR0L, PREINC1                                    \
      movf    PREINC1, 1, 0                                     \
   _endasm                                                      \
                                                                \
   /* Store the SW stack addr. */                               \
   _asm                                                         \
      movff   stack_low, FSR0L                                  \
      movff   stack_high, FSR0H                                 \
      movff   FSR1L, POSTINC0                                   \
      movff   FSR1H, POSTINC0                                   \
   _endasm                                                      \
}
```

_sched – Part 1

```
_sched
   GLOBAL  _sched
   #IFDEF  POSTTASKHOOK
      call    PostTaskHook
   #ENDIF
  … // skipped code here to select the next task
_restore_ctx
   GLOBAL  _restore_ctx

   movlb   0
   bsf     kernelState, 0          ; Change the kernel to USER mode
   locateTaskDescEntry
   locateStackAddrField
   loadNextAddrTo FSR0L, FSR0H        ; Extract task's stack addr
   loadNextAddrTo startAddressL, startAddressH
                                 ; Extract task's code addr
; Go chech whether the stack overflow occurred
   goto    _checkPanic
```

Steps 3 … 6

Hint: Interrupts still disabled.

_sched – Part 2

```
; If the stack remains intact, restore the task's context
_restore_now
    GLOBAL  _restore_now
    movlb   0
    movff   POSTDEC1, temp
    movff   POSTDEC1, temp              ; Extract # of H/W stack entries
    clrf    STKPTR                      ; backed up previously

… // skipped a section here
```

_sched – Part 3

```
restoreNextTmpdataByte
   movff   POSTDEC1, POSTDEC0        ; Restore .tmpdata + MATH_DATA
   movf    FSR0L, w                  ; section
   btfss   STATUS, N
   bra     restoreNextTmpdataByte

   movff   POSTDEC1, PRODL           ; Restore the rest of SFRs saved
   movff   POSTDEC1, PRODH           ; in previously task swapping out
   movff   POSTDEC1, TABLAT
   movff   POSTDEC1, TBLPTRL
   movff   POSTDEC1, TBLPTRH
   movff   POSTDEC1, TBLPTRU
   movff   POSTDEC1, FSR0H
   movff   POSTDEC1, FSR0L
   movff   POSTDEC1, FSR2H
   movff   POSTDEC1, FSR2L
   movff   POSTDEC1, BSR
   movff   POSTDEC1, WREG
   movff   POSTDEC1, STATUS

   bsf     INTCON, 6                 ; Enable OS/low prior. interrupt
   retfie                            ; Exit to where TOS pointed at
```

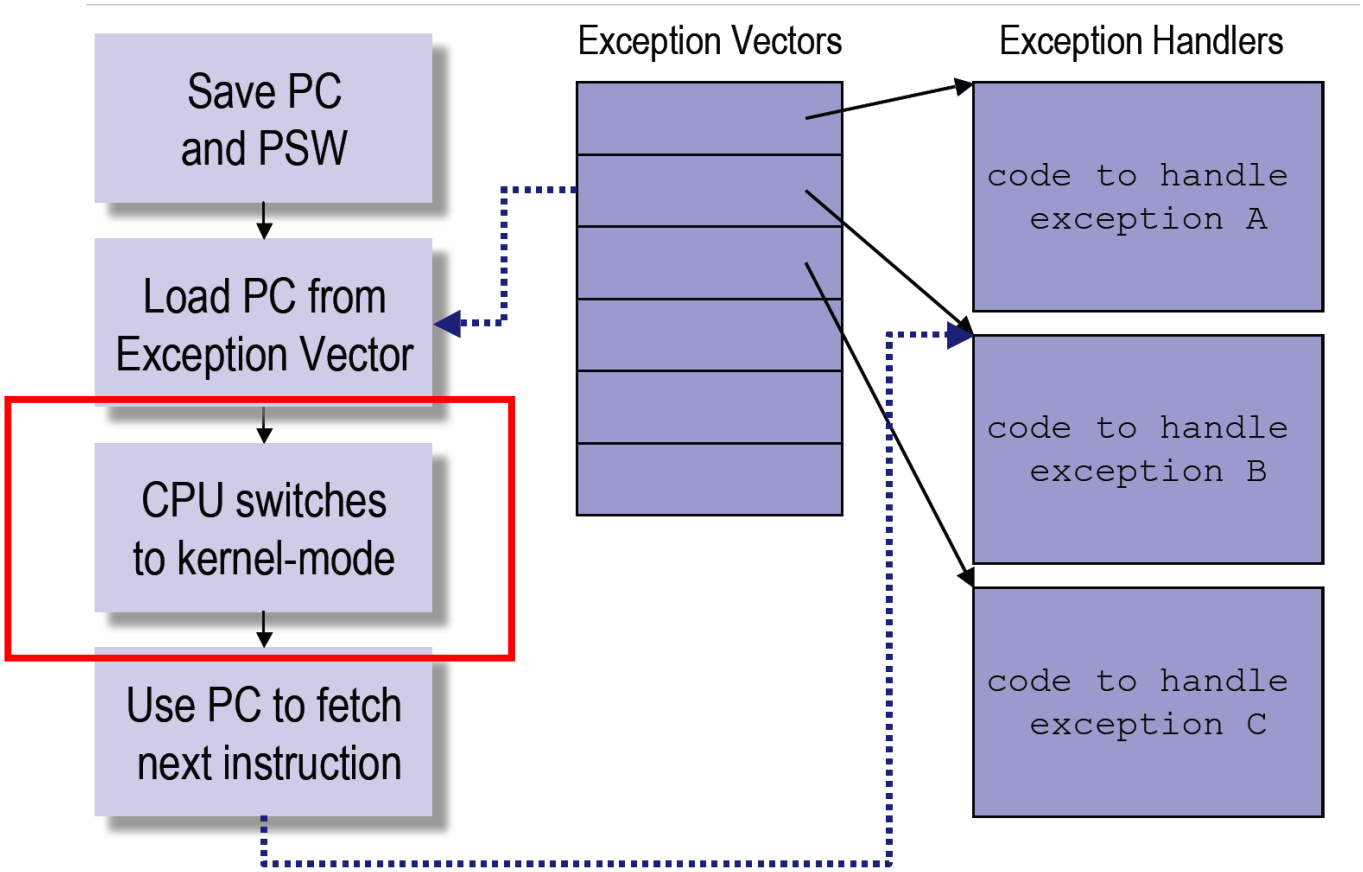
Q1: Anybody notices anything about the this process (up to now)?

Q2: How about starting process for the first time?

Resumes proc. execution.

EXECUTION OF THE OPERATING SYSTEM

Exception Handling



Divide-by-zero, invalid instruction, ...

User Exception Handling