

Chapter 6

Process Management

Introduction

- Scenario
 - One process running
 - One/more process performing I/O
 - One/more process waiting on resources
- Most of the complexity stems from the need to manage multiple processes

Introduction

- Process Manager
 - CPU sharing
 - Process synchronization
 - Deadlock prevention
- Each process has a Process Descriptor
 - Describes complete environment for a process

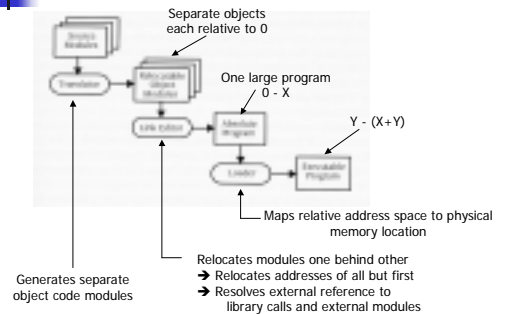
Process Descriptor

FIELD	DESCRIPTION
Internal process table	All internal tables of the process, such as its buffers or table index, point to this operating system code.
State	The process's current state.
Owner	A pointer to an account (identified by the owner's selected identification such as the login name). The descriptor contains a field for storing the owner identification.
Parent process identifier	A pointer to the process descriptor of the process's parent.
List of child process descriptors	A pointer to a list of other child processes of the process.
List of resources	A pointer to a list of resource resource types held by the process. Each resource type will be a descriptor of the number of units of the resource.
List of semaphore resources	Pointer to the semaphore resource list page (see Section 6.1.1.1).
List of the descriptors	A special case of the resource resource list.
Message queue	A special case of the semaphore resource list.
Permissions domain	A description of the access rights currently held by the process (see Chapter 1.4).
CPU status register address	A table of each of the CPU status registers at the last time the process owned the running state.
CPU general register address	A table of each of the CPU general registers at the last time the process owned the running state.

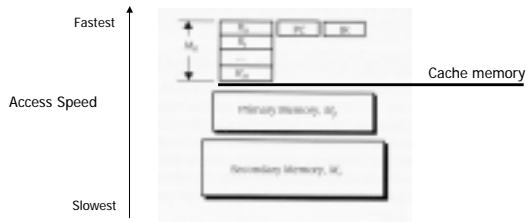
Process Address Space

- Defines all aspects of process computation
 - Program
 - Variables
 - ...
- Address space is generated/defined by translation

Creating an executable program



Basic Memory Hierarchy

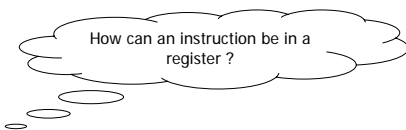


Basic Memory Hierarchy...

- At any point in the same program, element can be in
 - Secondary memory M_S
 - Primary memory M_P
 - Registers M_R
- Consistency is a Problem
 - M_S ≠ M_P ≠ M_R (code vs data)
 - When does one make them consistent ?
 - How ?

Consistency Problem

- Scheduler switching out processes – Context Switch
- Is Instruction a Problem ???
 - NO
 - Instructions are never modified
 - Separate Instruction and Data space
 - Therefore, M_{R_I} = M_{P_I} = M_{S_I}



Consistency Problem...

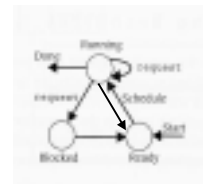
- Is Data a Problem ???
 - YES
 - Variable temporarily stored in register has value added to it
 - Therefore, M_{R_D} ≠ M_{P_D}
- On context switch, all registers are saved
 - Therefore, current state is saved

Sample Scenario...

- Suppose 'MOV X Y' instruction is executed
 - M_{R_I} ≠ M_{S_I}
- On context switch, is all of a process' memory flushed to M_S ?
 - No, only on page swap
- Hence, env_{process} = (M_R + M_S) + (...)
- Note:
 - Flushing of memory frees it up for incoming process => Page Swap

Process States

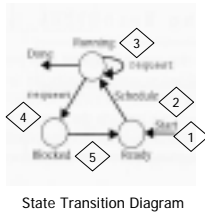
- Focus on Resource Management & Process Management
- Recall also that part of the process environment is its *state*



State Transition Diagram

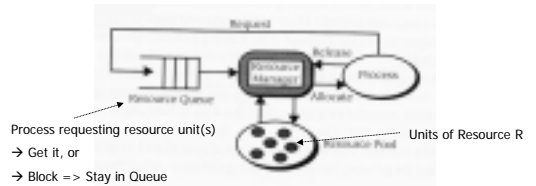
Process States...

- 1 When process enters 'Ready' state, it must compete for CPU. Memory has already been allocated
- 2 Process has CPU
- 3 Process requests resource that is immediately available → NO blocking
- 4 Process requests resource that is NOT yet available
- 5 Resource allocated, memory re-allocated?



Resources & Resource Manager

- 2 types of Resources
 - Reusable (Memory)
 - Consumable (Input/Time slice)

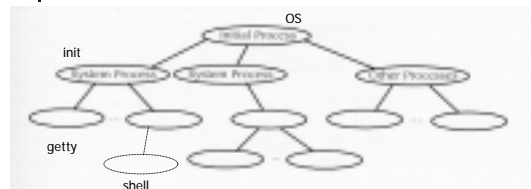


Resource Descriptor

- Each Resource R has a Resource Descriptor associated with it (similar to the process)
 => there is a "Status" for that Resource, and
 => a Resource Manager to manage it

FIELD	DESCRIPTION
Initial resource name	An internal name for the resource used by the operating system code. /dev/...
Total units	The number of units of this resource type managed into the system. 6
Available units	The number of units currently available. 3
Set of available units	The set of available units of this resource type that are available for use by processes. A, B, C
Set of blocked processes	The set of processes that have a pending request for units of this resource type. Only if * = 0

Process Hierarchy



- Conceptually, this is the way in which we would like to view it
- Root controls all processes i.e. Parent

Creating Processes

- Parent Process needs ability to
 - Block child
 - Activate child
 - Destroy child
 - Allocate resources to child
- True for User processes spawning child
- True for OS spawning init, getty, etc.
- Process hierarchy a natural, if fork/exec commands exist

UNIX fork command

- FORKUNIX
 - Shares text
 - Shares memory
 - Has its own address space
 - Cannot communicate with parent by referring variable stored in code
- Earlier definition: FORKCONWAY
 - Shares text
 - Shares resources
 - Shares address space
 - Process can communicate thru variables declared in code

Cooperating Processes

```

Prog
x, y : int
Proc A
  ref x & y
Proc B
  ref x & y
Fork "A"
Fork "B"

proc_A()
while(TRUE) {
  compute section A1();
  update(x);
  compute section A2();
  getNew(y);
}

proc_B()
while(TRUE) {
  retrieve(x);
  compute section B1();
  update(y);
  compute section B2();
}
    
```

Now processes A & B, share address space & can communicate thru declared variables

Problem ???

A can write 2 times before B reads

Synchronizing Access to Shared Variables

- Shared address space allows communication through declared variables automatically
- How then, can we synchronize access to them?
- Need Synchronization Primitives

=> JOIN & QUIT

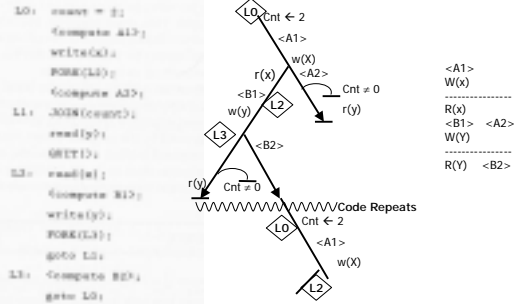
```

Prog
x, y : int
Proc A
  ref x & y
Proc B
  ref x & y
Fork "A"
Fork "B"
    
```

Fork, Join & Quit - Conway

- In addition to the "Fork(proc)" command, Conway also defined system calls to support process synchronization
- Join (count)
 - Un-interruptable
 - Decrement count; if count \neq 0 then Quit, else Continue
- Quit
 - Terminate process

Fork, Join, Quit example



A Simple Parent Program (Revisit)

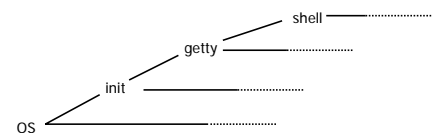
```

#include <sys/types.h>
#include <unistd.h>

int main (void)
{
  if (fork() == 0) { /* This is the child process */
    execve("/bin/sh", NULL, NULL);
    exit(0); /* Should never get here, terminate */
  }
  /* Parent code here */
  printf("Process[%d]: Parent is executing ...'\n", getpid());
  sleep(1);
  if (wait(&WALL) > 0) { /* Child terminating */
    printf("Process[%d]: Parent detects terminating child '\n",
           getpid());
    printf("Process[%d]: Parent terminating ...'\n", getpid());
  }
}
    
```

Spawning A Child Different From Parent

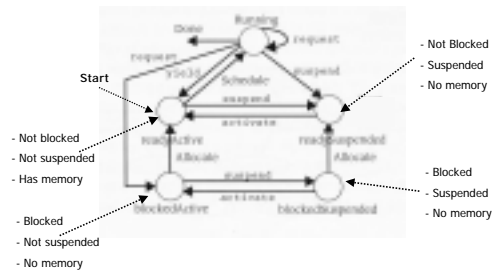
- Suppose we wish to spawn a child that is different from the parent
 - fork
 - execve(...)
- OS \rightarrow init \rightarrow getty \rightarrow shell



Factoring in additional Control Complexities

- Recall:
 - A parent process can suspend a child process
- Therefore, if a child is in run state and goes to ready (time slice up), and the parent runs and decides to suspend the child, then how do we reflect this in the process state diagram ???
- We need 2 more states
 - Ready suspended
 - Blocked suspended

Process State diagram reflecting Control



Give it a thought...

Why can a process NOT go from 'Ready Active' to 'Blocked Active' or 'Blocked Suspended' ?