

CS241 System Programming Process

Klara Nahrstedt

Lecture 3

1/25/06

Overview

- Process Concept
- Process Structure
- Process Operations
- Summary

Administrative

- Read T: Chapter 2.1
- Read R&R: Chapter 3

Processes

- A **process** is an abstraction for sequence of operations that implement a computation. A process may be manipulated, suspended, scheduled and terminated
- Process is a unit of work in a modern computer
- Process Types:
 - OS processes executing system code
 - User processes executing user code
- Processes are executed concurrently with CPU multiplexing among them

Process Management

- Process is also used to describe concurrent user activities and many forms of parallel computation
- A process is one of the key concepts underlying modern operating system design

An Abstraction

- In a time-sharing system, the actual execution of the instructions of a process may become interleaved with other instructions of other processes
- In a multiprocessor system, the concurrent processes may each execute on an independent processor

Thread of Control

- The path that a computation taken through a program corresponding to the individual statements that are executed by a particular process

Locus of Control

- The end of the thread of control of a process marking the instruction that will be executed next

Concurrent Program

- A program that has many processes

Discussion

- What mechanisms are needed to implement a process?
- How does hardware architecture support the concept of a process?

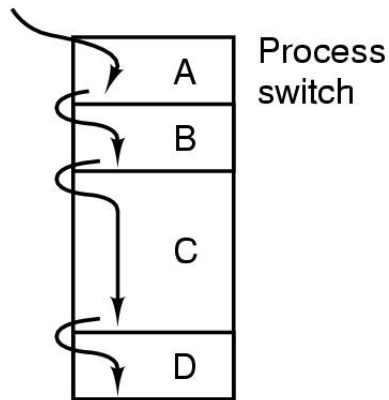
Motivation for a Process

- Process
 - Each process has an ``abstract" locus of control
 - Any exchanges of control are done automatically and implicitly
 - Runs in parallel on multiprocessor

Processes

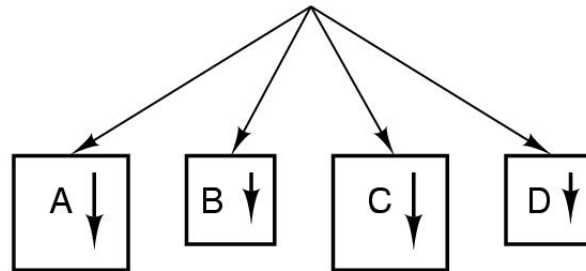
The Multiprogramming Process Model

One program counter

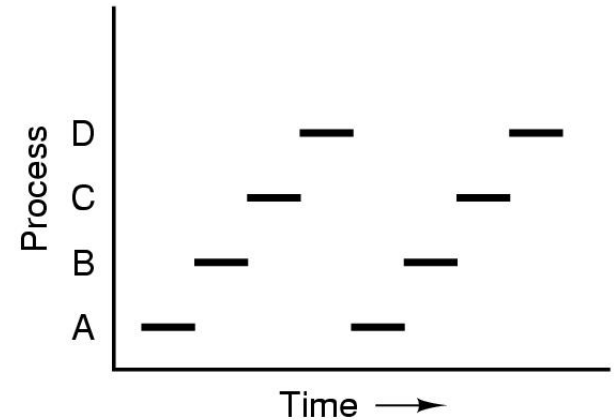


(a)

Four program counters



(b)



(c)

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

Process Identification

- UNIX identifies processes via unique value
 - Process ID
 - Each process has also parent process ID since each process is created from a parent process.
 - Root process is the 'init' process
- '*getpid*' and '*getppid*' – functions to return process ID (PID) and parent process ID (PPID)

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
int main (void) {
```

```
    printf("I am process %ld\n", (long)getpid());
```

```
    printf("My parent id %ld\n", (long)getppid());
```

```
    return 0;
```

```
}
```

Process Creation

- System initialization
 - reboot
- Execution of a process creation system call
 - Fork() :
#include <unistd.h>
pid_t fork(void);
- User request to create a new process
 - Command line or click an icon
- Initiation of a batch job
 - Cron
- Processes are created and deleted dynamically

Creating a Process in Unix

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int x;

    x = 0;
    fork();
    x = 1;
    printf("I am process %ld and my x is %d\n", (long)getpid(), x);
    return 0;
}
```

UNIX Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    pid_t parentpid;
    pid_t childpid;

    if ((childpid = fork()) == -1) {
        perror("can't create a new process");
        exit(1);
    }
    else if (childpid == 0) { /* child process executes */
        printf("child: childpid = %d, parentpid = %d \n", getpid(), getppid());
        exit(0);
    }
    else { /*parent process executes */
        printf("parent: childpid = %d, parentpid = %d \n", childpid, getpid());
        exit(0);
    }
}
```

Process Operations (Creation)

- When creating a process, we need resources such as CPU, memory files, I/O devices
 - When creating a new process, process can get resources from the OS or from the parent process
 - When getting resources from a parent, this means that the child process is restricted to a subset of parent resources
 - Prevents many processes from overloading system
 - When creating a new process , execution possibilities are
 - Parent continues concurrently with child
 - Parent waits until child has terminated
 - When creating a new process, address space possibilities are:
 - Child process is duplicate of parent process
 - Child process had a program loaded into it

Process Termination

- Normal exit (voluntary)
 - End of main()
- Error exit (voluntary)
 - `exit(2)`
#include <stdlib.h>
void exit(int status);
- Fatal error (involuntary)
 - Divide by 0, core dump
- Killed by another process (involuntary)
 - Kill `proclD`, end task

Process Operations (Termination)

- When a process finishes last statement, it asks OS to delete it
- Child process may return output to parent process, and all child's resources are de-allocated.
- Other termination possibilities
 - Abort by parent process invoked
 - Child has exceeded its usage of some resources
 - Task assigned to child is no longer required
 - Parent is exiting and OS does not allow child to continue without parent

Process Hierarchies

- Parent creates a child process, a child process can create its own processes
- Forms a hierarchy
 - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
 - all processes are created equal

Wait Function

- *wait* function allows parent process to wait (block) until child finishes
- *wait* function causes the caller to suspend execution until child's status is available
- *waitpid* function allows a parent to wait for a particular child

| errno | cause |
|--------|---|
| ECHILD | Caller has no unwaited-for children |
| EINTR | Function was interrupted by signal |
| EINVAL | Options parameter of <i>waitpid</i> was invalid |

Waiting for a child to finish

```
#include <errno.h>
#include <sys/wait.h>

pid_t childpid;

childpid = wait(NULL);
if (childpid != 0)
    printf("waited for child with pid %ld\n", childpid):
```

Status Values

The following macros are available for checking the return status of a child:

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

```
WIFEXITED(int stat_val)  
WEXITSTATUS(int stat_val)  
WIFSIGNALED(int stat_val)  
WTERMSIG(int stat_val)  
WIFSTOPPED(int stat_val)  
WSTOPSIG(int stat_val)
```

They are used in pairs.

If `WIFEXITED` returns true, the child executed normally and the return status (at most 8 bits) can be gotten with `WEXITSTATUS`.

Typical process creation code

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void) {
    pid_t childpid;
    /* set up signal handlers here ... */
    childpid = fork();
    if (childpid == -1) {
        perror("Failed to fork");
        return 1;
    }
    if (childpid == 0)
        fprintf(stderr, "I am child %ld\n", (long)getpid());
    else if (wait(NULL) != childpid)
        fprintf(stderr, "A signal must have interrupted the wait!\n");
    else
        fprintf(stderr, "I am parent %ld with child %ld\n", (long)getpid(),
            (long)childpid);
    return 0;
}
```

Exec Function

- Exec function allows child process to execute code that is different from that of parent
- Exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
- Exec functions return -1 and set errno if unsuccessful

Exec Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void) {
    pid_t childpid;

    childpid = fork();
    if (childpid == -1) {
        perror("Failed to fork");
        return 1;
    }
    if (childpid == 0) { /*child code*/
        execl("/bin/ls", "ls", "-l", NULL);
        perror("Child failed to exec ls");
        return 1;
    }
    if (wait(NULL) != childpid) { /* parent code */
        perror("Parent failed to wait due to signal or error");
        return 1;
    }
    return 0;
}
```

Background Processes and Daemons

- **Shell** – command interpreter creates background processes if line ends with &
 - When shell creates a background process, it does not wait for the process to complete before issuing a prompt and accepting another command
 - Ctrl-C does not terminate background process
- **Daemon** is a background process that normally runs indefinitely
- Example: mail client, file transfer, pageout daemon

Summary

- Process – an executable program – important abstraction in OS
- Process Operations
 - Creation of processes
 - Termination of processes
 - Wait
 - Exec
- Read T: Chapter 2.1
- Read R&R Chapter 3