Review

- An OS in action
- Processes and Programs
- A day in the life of a process
- CPU scheduling
  - pre-emptive vs non pre-emptive
  - FCFS
  - Round Robin

Outline

- Finish scheduling
- Process Manipulation
- Multithreaded Programming

Preview

- Threads and Multithreading

STCF and SRTCF

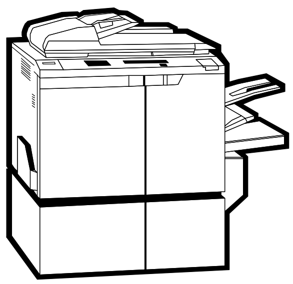
STCF: Shortest Time to Completion First

- Scheduler implements a queue sorted by amount of time to run the process
- Process at the head of queue is allowed to run *to completion* before bringing in the next process

SRTCF: Shortest Remaining Time to Completion First

- Same as STCF
- only, jobs can be preempted

The Idea



- Get short jobs out of the system quickly
- Big effect on short jobs, small effects on large jobs
- Get better average response time

In fact, these protocols are *optimal*!

The Catch

- Unfair
  - constant stream of short jobs can arbitrarily delay long jobs
- Clairvoyant
  - need to know the future!
  - easy: ask the user
  - yeah, right!

What if you don't subscribe to the Psychic Network?

Use past to predict future!

– e.g., if program was I/O bound in the past, likely to be I/O bound in the future

– used in virtual memory, file system, CPU scheduling…

t_n = length of last CPU burst

τ_{n+1} = predicted value of next CPU burst

$$\boxed{\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n}$$

Multi-Feedback Queues (UNIX)

- Scheduler implements several RR queues such that the processes in one queue all have the same priority
- Process at the head of the highest priority queue is allowed to run until:
  - a time quantum expires, in which case process re-enters queue
  - process enters the wait state, in which process is out of the queue and re-enters when the wait condition is no longer relevant
- After running for a while, a process is relegated to the next queue in priority order (its priority is decreased)
- After spending time in the I/O wait, a process is promoted to the highest priority queue (its priority is set to max)

Can we fool the protocol?

Process Manipulation in UNIX

Basic process manipulation: creation, program loading, exiting, …

- fork(), exec(), wait(), exit()

Process signaling

- kill()

Process control

- ptrace(), nice(), sleep()

An example application: The UNIX shell

Process Creation

- The system creates the first process (*sysproc* in UNIX)
- The first process creates other processes such that:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships can be expressed by a process tree
- For example, in UNIX the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (num of processes, priorities…
- The system interface includes a call to create processes
  - In UNIX, for example, it is called *fork*()

Example: UNIX's fork()

- Creates a child process such that it inherits:
  - identical copies of all parent's variables & memory
  - identical copies of all parent's CPU registers (except one)
- Both parent and child execute at the same point after fork() returns:
  - for the child, fork() returns 0
  - for the parent, fork() returns the process identifier of the child
- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - *expensive*!

Usage of fork()

In the parent process:

```
main()
…
int pid = fork();            // create a child
if(pid == 0) {               // child continues here
…
}
else {                       // parent continues here
…
}
```

Characteristics of fork()

- Parent and child's code must be in the same "program"
  - limits the usefulness for general purpose process creation
  - gets out of control quickly if many children are to be created!
- The child inherits all the open files & network connections
  - having parent and child share the I/O & network is tricky
  - behavior with respect to network connections vary
  - if the child does not use the inherited open files, then the work to set up this sharing is useless, and resources are consumed within the OS for no good reason
- Again, and lest you forget, it is expensive

Program Loading: exec()

- The exec() call allows a process to "load" a different program and start execution at _start
- It allows a process to specify the number of arguments (argc) and the string argument array (argv)
- It the call is successful
  - it is the same process!!
  - it runs a different program!!
- Two implementation options:
  - overwrite current memory segments with the new values
  - allocate new memory segments, load them with the new values, and deallocate old segments

General Purpose Process Creation

In the parent process:

```
main()
…
int pid = fork();                // create a child
if(pid == 0) {                   // child continues here
    exec("program", argc, argv0, argv1, …);
}
else {                           // parent continues here
…
}
```

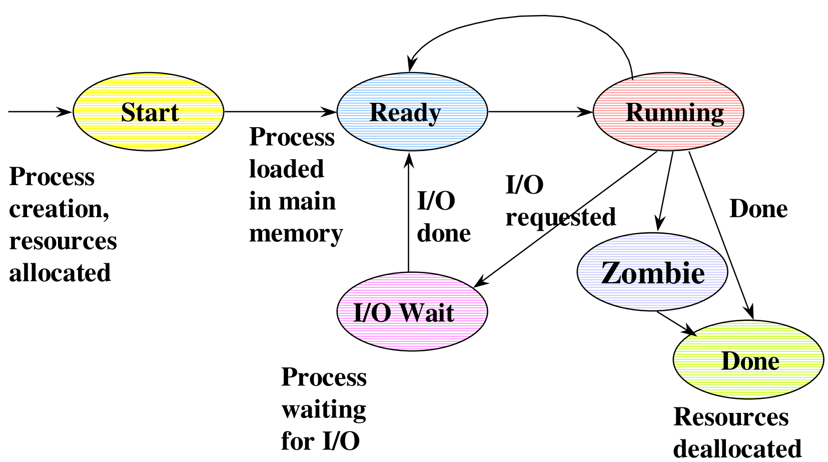
Properties of the fork/exec sequence

- In 99% of the time, we call exec() after calling fork()
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
  - might as well combine them in one call (OS/2)
- vfork()
  - a system call that creates a process "without" creating an identical memory image
  - sometimes called lightweight fork()
  - child process is understood to call exec() almost immediately

Orderly Termination: exit()

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the "result" of the program as an argument (CRT0 calls exit() with the value returned by the main program)
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies

The Process Life Cycle



The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The wait() system call serves this purpose
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Process Control

O.S. must include calls to enable special control of a process:

- Priority manipulation:
  - nice(), which specifies base process priority (initial priority)
  - but recall, priority in UNIX decays as the process consumes CPU
- Debugging support:
  - ptrace(), allows a process to be put under control of another
  - the other process can set breakpoints, examine registers, etc.
- Alarms and time:
  - sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality

Signaling

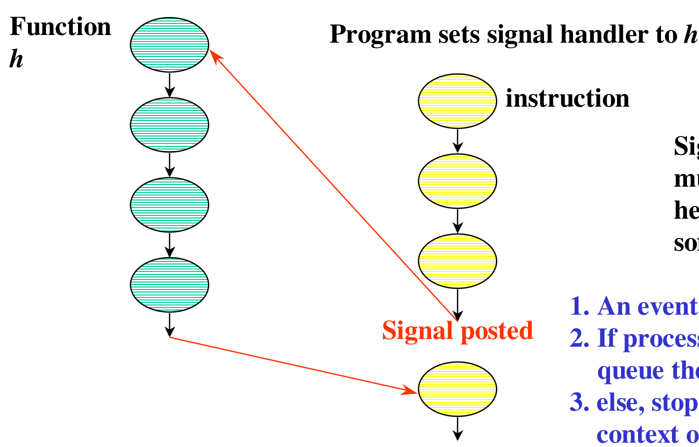
- The O.S. translates some events into *asynchronous* signals:
  - input/output
  - alarms
- The O.S. also translates exceptions into signals, e.g.:
  - Memory access violations, illegal instructions, overflow, etc.
- Signals also used for process control, e.g.:
  - SIGKILL to kill a process, SIGSTOP to stop a process, etc.
- Rudimentary process communications, e.g.:
  - processes may use the *kill*() system call to send signals to each other (SIGUSR1, SIGUSR2)

Signal Handling

- A process can:
  - rely on the default signal handler (e.g. core dump in UNIX)
  - ignore signal (temporarily or permanently, dangerous)
  - sets its own handler
- Fundamental difference between signals & regular I/O
  - A process gets regular I/O when it chooses to do so (synchronous)
  - A process is interrupted and forced to handle a signal whenever a signal is posted (and not ignored). This is asynchronous

Signals are very difficult to deal with and are better avoided

How Does it Work?



Signals work very much like interrupts hence the name software interrupts

1. **An event of interest occurs**
2. **If process masks event out queue the signal**
3. **else, stop process, push its context on stack, force process to jump to handler**

Tying it All Together: The Shell

```
while(! EOF) {
read input
handle regular expressions
if(built-in command) {
    execute command; continue; }
if(child = fork()) {                    // create a child to run command
    child = wait(&res);
}
else {                                  // command is run here
    exec(command, argc, argv0, argv1, … env);
}
```

Other Process Control by Shell

- Translates <CTRL-C> to kill command using the kill() system call with SIGKILL
- Translates <CTRL-Z> to stop the foreground process using the kill() system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later