# 3. Process Concept

3.1 Describe the differences among short-term, medium-term, and longterm scheduling.

Answer:

Short-term (CPU scheduler): selects a process from those that are in memory and ready to execute, and and allocates the CPU to it.

Medium-term (memory manager): selects processes from the ready or blocked queue and removes them from memory, then reinstates them later to continue running.

Long-term (job scheduler): determines which jobs are brought into the system for processing.

3.2 Describe the actions taken by a kernel to switch context

(a) Among threads

(b) Among processes

Answer:

The thread context (registers, PC, and stack, plus accounting info if appropriate) must be saved and another thread's context must be loaded.

The same as (a), except that the memory and resource info must also be stored and that for the next process must be loaded.

3.4 Explain the role of the init process on UNIX and Linux systems in regard to process termination.

Answer:

When a process is terminated, it briefly moves to the zombie state and remains in that state until the parent invokes a call to wait(). When this occurs, the process id as well as entry in the process table are both released. However, if a parent does not invoke wait(), the child process remains a zombie as long as the parent remains alive. Once the parent process terminates, the init process becomes the new parent of the zombie. Periodically, the init process calls wait() which ultimately releases the pid and entry in the process table of the zombie process.

3.5 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

Answer:

16

3.6 Explain the circumstances under which the line of code marked printf("Line J" ) in Figure 3.33 will be reached.

```
int main()
{
pid_t pid;

//fork a child process
pid = fork();

if(pid<0) //error occurred
{
fprintf(stderr, "fork failed);
return 1;
}
else if(pid == 0) //child process
{
execlp("/bin/ls", "ls", NULL)
printf("LINE J");
}

else //parent process
{
wait(NULL);
printf("child complete");
}
return 0;
}
```

Answer:

The call to exec() replaces the address space of the process with the program specified as the parameter to exec(). If the call to exec() succeeds, the new program is now running and control from the call to exec() never returns. In this scenario, the line printf("Line J"); would never be performed. However, if an error occurs in the call to exec(), the function returns control and therefor the line printf("Line J"); would be performed.

3.7 Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid, pid1;
/ fork a child process /
pid = fork();
if (pid < 0) { / error occurred /
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { / child process /
pid1 = getpid();
printf("child: pid = %d",pid); / A /
printf("child: pid1 = %d",pid1); / B /
}
else { / parent process /
pid1 = getpid();
printf("parent: pid = %d",pid); / C /
printf("parent: pid1 = %d",pid1); / D /
wait(NULL);
}
return 0;
}
```

Answer:

A = 0

B = 2603

C = 2603

D = 2600

3.8 Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

Answer:

Simple communication works well with ordinary pipes. An ordinary pipe can be used where the producer writes the file to the pipe and the consumer reads the files and counts the number of characters in the file. Next, for an example where named pipes are more suitable, consider the situation where several processes may write messages to a log. When processes wish to write a message to the log, they write it to the named pipe. A server reads the messages from the named pipe and writes them to the log file.

3.9 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

Answer:

If an RPC mechanism cannot support either the "at most once" or "at least once" semantics, then the RPC server cannot guarantee that a remote procedure will not be invoked multiple occurrences. Consider if a remote procedure were withdrawing money from a bank account on a system that did not support these semantics. It is possible that a single invocation of the remote procedure might lead to multiple withdrawals on the server. For a system to support either of these semantics generally requires the server maintain some form of client state such as the timestamp described in the text. If a system were unable to support either of these semantics, then such a system could only safely provide remote procedures that do not alter data or provide time-sensitive results. Using our bank account as an example, we certainly require "at most once" or "at least once" semantics for performing a withdrawal (or deposit!). However, an inquiry into an account balance or other account information such as name, address, etc. does not require these semantics.

3.10 Using the program shown in Figure 3.35, explain what the output will be at lines X and Y.

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

#define SIZE 5
int sums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

pid = fork();

if (pid == 0)
{
for(i=0; i <SIZE; i++)
{
nums[i] *= -i;
printf("CHILD %d ", nums[i] ); // LINE X
}
}
else if (pid > 0)
{
wait(NULL);
for(i=0; i < SIZE; i++)
printf("PARENT: %d ", nums[i] ); // LINE Y
}

return 0;
}
```

Answer:
Because the child is a copy of the parent, any changes the child makes will occur in its copy of the data and won't be reflected in the parent. As a result, the values output by the child at line X are 0, -1, -4, -9, -16. The values output by the parent at line Y are 0, 1, 2, 3, 4

3.11 What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.
a. Synchronous and asynchronous communication
b. Automatic and explicit buffering
c. Send by copy and send by reference
d. Fixed-sized and variable-sized messages
Answer:
a)
A benefit of synchronous communication is that it allows a rendezvous between the sender and receiver. A disadvantage of a blocking send is that a rendezvous may not be required and the message could be delivered asynchronously. As a result, message-passing systems often provide both forms of synchronization.

b)
Automatic buffering provides a queue with indefinite length, thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications on how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely that memory will be wasted with explicit buffering.

c)
Send by copy does not allow the receiver to alter the state of the parameter; send by reference does allow it. A benefit of send by reference is that it allows the programmer to write a distributed version of a centralized application. Java's RMI provides both; however, passing a parameter by reference requires declaring the parameter as a remote object as well. Send by copy may increase safety because the value rather than a reference to the location of the value is passed and therefore the original cannot be corrupted. However, if the value is something large like a struct or binary object, it is advantageous to pass a reference to keep the stack smaller. Also, it is sometimes preferred or necessary to change the original value in-place which requires send by

reference.

d)
The implications of this are mostly related to buffering issues; with fixed-size messages, a buffer with a specific size can hold a known number of messages.
The number of variable-sized messages that can be held by such a buffer is unknown.
Consider how Windows 2000 handles this situation: with fixed-sized messages (anything < 256 bytes), the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages (i.e. variable-sized messages) use shared memory to pass the message.

Fixed-size messages are easier to implement (for the kernel) but multiple messages may need to be sent if they are too long. This requires a lot of overhead and preparation (by the user program) so variable-sized messages may be better in some cases.