Processes and Threads

Tom Kelliher, CS 311

Feb. 20, 2012

Announcements:

Exam in one week.

From last time:

1. Adding a syscall to the kernel.

Outline:

1. Syscall assignment.
2. Processes.
3. Threads.

1 Process Resources

1. *Program in execution*.
2. Serial, ordered execution within a single process. (Contrast *task* with multiple *threads*.)
3. "Parallel" unordered execution between processes.

Three issues to address

1. Specification and implementation of processes — the issue of **concurrency** (raises the issue of the *primitive* operations).

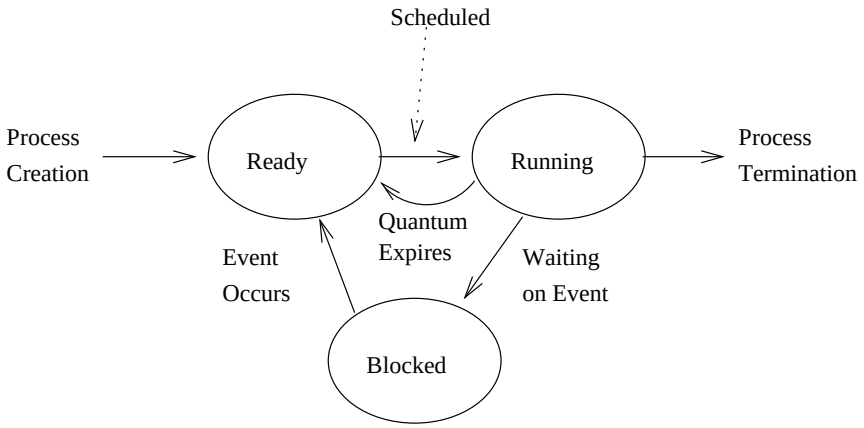
2. Resolution of **competition** for resources: CPU, memory, I/O devices, etc.

3. Provision for **communication** between processes.

1.1 Process States

Three possible states for a process:

- Running — currently being executed by the processor
- Ready — waiting to get the processor
- Blocked (waiting) — waiting for an event to occur: I/O completion, signal, etc. (Suspended — ready to run but not eligible.)



How many in each state?

1.2 A Process' Resources

Kept in a process control block (PCB) for each process:

1. Code (possibly shared among processes).
2. Execution stack — stack frames.
3. CPU state — general purpose registers, PC, status register, etc.
4. Heap — dynamically allocated storage.
5. State — running, ready, blocked, zombie, etc.
6. Scheduling information — priority, total CPU time, wall time, last burst, etc.
7. Memory management — page, segment tables.
8. I/O status — devices allocated, open files, pending I/O requests, postponed resource requests (deadlock avoidance).
9. Accounting — owner, CPU time, disk usage, parent, child processes, etc.

Contrast *program*.

PCB updated during context switches (kernel in control).

Should a process be able to manipulate its PCB?

2 Process Scheduling

Determination of which process to run next (CPU scheduling).

Multiple queues for holding processes:

1. Ready queue — priority order.
2. I/O queues — request order.

   Consider a disk write:

   (a) Syscall.

(b) Schedule the write.

(c) Modify PCB state, move to I/O queue.

(d) Call short term scheduler to perform context switch.

Is it necessary to wait on a disk *write*?

3. Event queues — waiting on child completion, sleeping on timer, waiting for request (`inetd`).

Three types of schedulers:

- Long term scheduler.
- Medium term scheduler.
- Short term (CPU) scheduler.

2.1 Long Term Scheduler

Determines overall job mix:

1. Balance of I/O, CPU bound jobs.
2. Attempts to maximize CPU utilization, throughput, or some other measure.
3. Runs infrequently.

2.2 Medium Term Scheduler

Cleans up after poor long term scheduler decisions:

1. Over-committed memory — thrashing.

2. Determines candidate processes for suspending and paging out.

3. Decreases degree of multiprogramming.

4. Runs only when needed.

2.3 CPU Scheduler

Decides which process to run *next*:

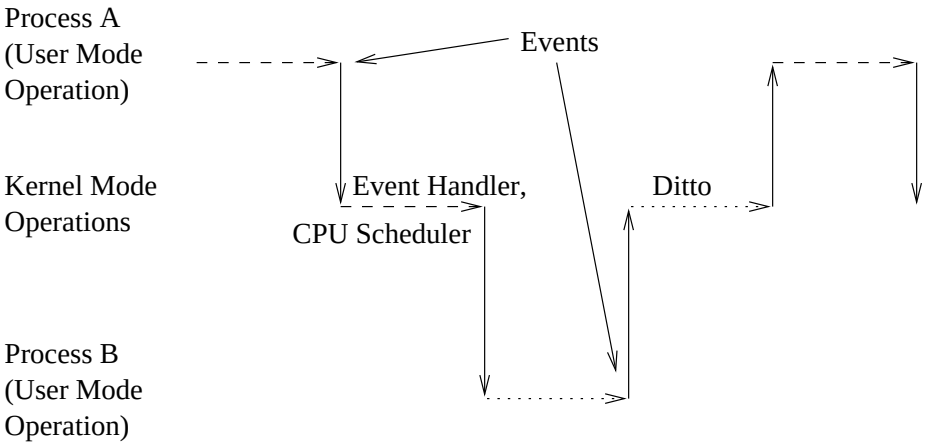
1. Picks among processes in ready queue.

2. Priority function.

3. Runs frequently — must be efficient.

3 Context Switching

Time line schematic:



4 Operations on Processes

4.1 Process Creation

Parent, child.

Where does the child's resources come from? By "resources" we mean:

1. Stack.
2. Heap.
3. Code.
4. Environment — environment variables, open files, devices, etc.

Design questions:

- New text, same text?
- Make a copy of the memory areas? (Expensive.)
- Copy the environment?
- How are open files handled?

Solutions to the "copy the parent's address space" problem:

1. Copy on write — Mark all parent's pages read only and shared by parent & child. On any attempted write to such a page, make a copy and assign it to child. Fix page tables.
2. `vfork` — No copying at all. It is assumed that child will perform an `exec`, which provides a private address space.

5 Threads

Heavyweight process — expensive context switch.

Thread:

1. Lightweight process.
2. Consist of PC, general purpose register state, stack.
3. Shares code, heap, resources with *peer* threads.
4. Easy context switches.

Task: peer threads, shared memory and resources.

Can peer threads scribble over each other?

What about non-peer threads?

User-level threads:

1. Implemented in user-level libraries; no system calls.
2. Kernel only knows about the task.
3. Threads schedule themselves within task.
4. Advantage: fast context switch.
5. Disadvantages:
   (a) Unbalanced kernel level scheduling.
   (b) If one thread blocks on a system call, peer threads are also blocked.

Kernel-level threads:

1. Kernel knows of individual threads.

2. Advantage: If a thread blocks, its peers can still proceed.

3. Disadvantage: Slower context switch (kernel involved).

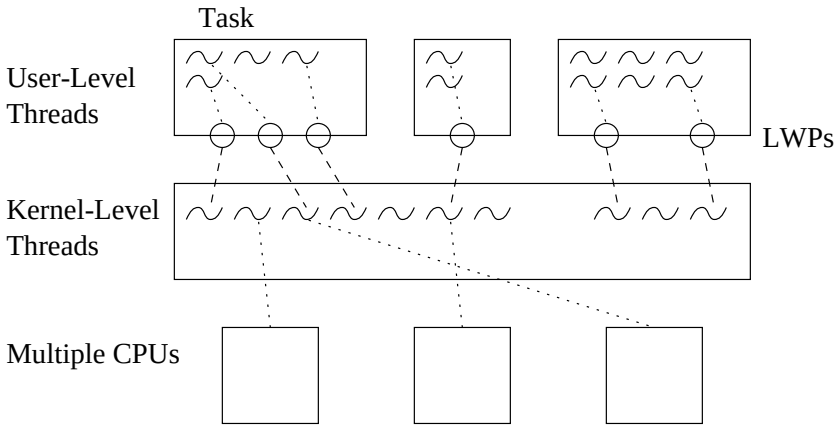
How do threads compare to processes?

1. Context switch time.

2. Shared data space. (Improved throughput for file server: shared data, quicker response.)

5.1 Example: Solaris 2

User-level threads multiplexed upon lightweight processes:



6 IPC Mechanisms

Basics: `send()`, `receive()` primitives.

Design Issues:

1. Link establishment mechanisms:

   (a) Direct or indirect naming.

   (b) Circuit or no circuit.

2. More than two processes per link (multicasting).

3. Link buffering:

   (a) Zero capacity.

   (b) Bounded capacity.

   (c) Infinite capacity.

4. Variable- or fixed-size messages.

5. Unidirectional or bidirectional links (symmetry).

6. Resolving lost messages.

7. Resolving out-of-order messages.

8. Resolving duplicated messages.

6.1 Mailboxes — An Indirect Communication Mechanism

Resources owned by kernel.

Messages kept in a queue.

Assume:

1. Only allocating process may execute receive.

2. Any process (including "owner") may send.

3. Variable-sized messages.

4. Infinite capacity.

Primitives:

1. `int AllocateMB(void)`
2. `int Send(int mb, char* message)`
3. `int Receive(int mb, char* message)`
4. `int FreeMB(int mb)`

6.1.1 Example: Process Synchronization

Consider:

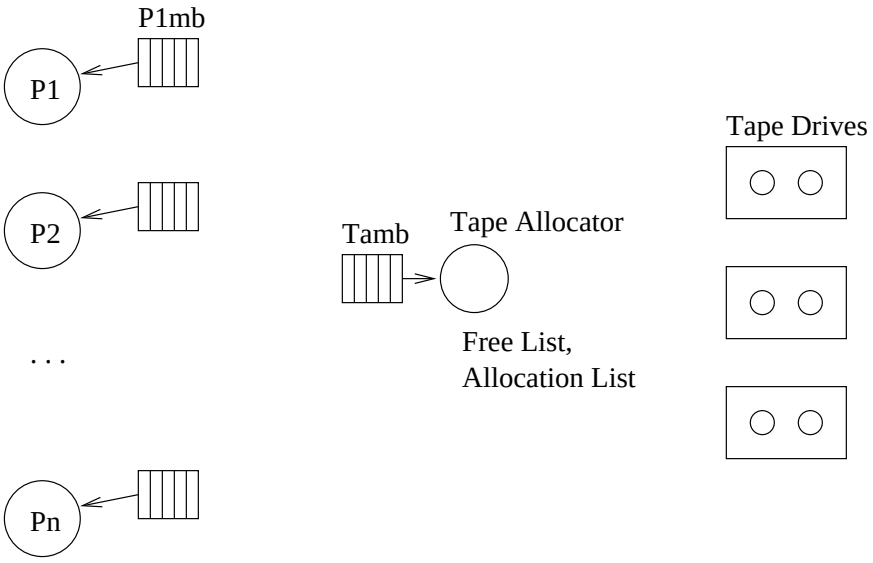
```
Process1()
{
   ...
   S1;
   ...
}

Process2()
{
   ...
   S2;
   ...
}
```

How can we guarantee that `S1` executes before `S2`?

6.1.2 Example: Tape Drive Allocation and Use

The situation:



Tape allocator process:

```
initialize();
while (1)
{
   Receive(Tamb, message);
   if (message is a request)

      if (there are enough tape drives)

         for each tape drive being allocated
         {
            fork a handler daemon;
            send daemon mb # in message to requesting process;
            update lists;
         }

      else

         send a rejection message;

   else if (message is a return)
   {
      update lists;
      send an ack message;
   }

   else
```

```
      ignore illegal messages;
}
```

Summary of user process actions:

1. Send request to tape allocator.
2. Receive message back giving mailbox(es) to use in communicating with tape drive(s).
3. Start sending/receiving with tape drive daemon(s).
4. Close tape drives.
5. Send message to tape allocator returning tape drive(s).